Under consideration for publication in Formal Aspects of Computing

Abstract Conjunctive Partial Deduction for the Analysis and Compilation of Coroutines

Vincent Nys and Danny De Schreye

KU Leuven, Department of Computer Science, Celestijnenlaan 200A, Heverlee, Belgium

Abstract. We provide an approach to formally analyze the computational behavior of coroutines in Logic Programs and to compile these computations into new programs, not requiring any support for coroutines. The problem was already studied near to 30 years ago, in an analysis and transformation technique called Compiling Control. However, this technique had a strong ad hoc flavor: the completeness of the analysis was not well understood and its symbolic evaluation was also rather ad hoc. We show how Abstract Conjunctive Partial Deduction, introduced by Leuschel in 2004, provides an appropriate setting to redefine Compiling Control. We define an abstract domain and all abstract operations required by Abstract Conjunctive Partial Deduction. We prove that these concepts satisfy all the correctness conditions imposed by the framework and therefore inherit its main correctness theorem. We also show that there exist more complex coroutining examples which do not fit within Abstract Conjunctive Partial Deduction and we propose some further extensions to include them.

Keywords: Coroutines, Compiling Control, Abstract Conjunctive Partial Deduction

1. Introduction

Coroutines are a powerful means of supporting complex computation flows. They can be very useful for improving the efficiency of declaratively written programs, in particular for generate-and-test based programs. On the other hand, obtaining a deep understanding of the computation flows underlying the coroutines is notoriously difficult.

In this paper we restrict our attention to pure, definite Logic Programs. In this context, the problem was already studied nearly 30 years ago. Bruynooghe et al. [1986] and Bruynooghe et al. [1989] present an analysis and transformation technique for coroutines, called Compiling Control (CC for short). The purpose of the CC transformation is the following: transform a given program, P, into a program P', so that a computation with P' under the standard selection rule mimics the computation with P under a non-standard selection

Correspondence and offprint requests to: Vincent Nys, Department of Computer Science, KU Leuven Celestijnenlaan 200A, B-3001, Heverlee, Belgium. e-mail: vincent.nys@kuleuven.be

rule. In particular, given a coroutining selection rule for a given Logic Program, the transformed program will execute the coroutining if it is evaluated under the standard selection rule of Prolog. This is also the aim of the current paper. In particular, using the proposed transformation, one could start with a program with delay declarations and the transformation produces a program without delay declarations, with the same runtime behavior.

To achieve this aim, CC consists of two phases: an analysis phase and a synthesis phase. The analysis phase analyzes the computations of a program for a given query pattern and under a (non-standard) selection rule. The query pattern is expressed in terms of a combination of type, mode and aliasing information. The selection rule is instantiation-based, meaning that different choices in atom selection need to be based on different instantiations of these atoms. The analysis results in what is called a "trace tree", which is a finite upper part of a symbolic execution tree that one can construct for the given query pattern, selection rule and program. In the synthesis phase, a finite number of clauses are generated, so that each clause synthesizes the computation in some branch of the trace tree and such that all computations in the trace tree have been synthesized by some clause. The technique was implemented, formalized and proven correct, under certain fairly technical conditions.

Unfortunately, the CC transformation has a rather ad hoc flavor. It was very hard to show that the analysis phase of the transformation was complete, in the sense that a sufficiently large part of the computation had been analyzed to be able to capture all concrete computations that could possibly occur at run time. Even the very idea of a "symbolic execution" had an ad hoc flavor. It seemed that it should be possible to see this as an instance of a more general framework for analysis of computations.

Fortunately, since the development of CC a number of important advances have been achieved in analysis and transformation:

- General frameworks for abstract interpretation (e.g. Bruynooghe [1991]) were developed. It is clear that abstract interpretation has the potential to provide a better setting for developing the CC analysis. But it still seems different, because abstract interpretation is most often used to analyze properties that hold during or after a computation, while in CC we are interested in analyzing the computational flow itself.
- Partial deduction of Logic Programs, originally introduced in Komorowski [1981], was further developed (e.g. in Gallagher [1986]). Partial deduction seems very similar to CC, but the exact relationship was never identified. When John Lloyd and John Shepherdson formalized the issues of correctness and completeness of partial deduction in Lloyd and Shepherdson [1991], this provided a new framework for thinking about a complete analysis of a computational behavior and it was clear that some variant of this could improve the CC analysis.
- Conjunctive partial deduction (see De Schreye et al. [1999]) seems even closer to CC. In an analysis for a CC transformation, one really does not want to split up the conjunctions of atoms into separate ones and then analyze the computations for these atoms separately. It is crucial that one can analyze the computation for certain atoms in conjunction (which is how conjunctive partial deduction generalizes partial deduction), so that their behavior under the non-standard selection rule may be observed. A similar idea has been applied in the context of Fold/Unfold transformations in De Angelis et al. [2015].
- Finally, abstract (conjunctive) partial deduction (Leuschel [2004]) brings all these features together. It provides an extension of (conjunctive) partial deduction in which the analysis is based on abstract interpretation, rather than on concrete evaluation.

In this paper we will demonstrate that abstract conjunctive partial deduction (ACPD for short) is indeed a suitable framework to redefine CC in such a way that the flaws of the original approach are overcome. We show that for simple problems in the CC context, ACPD can produce the transformation automatically. We also show that for more complex CC transformations, ACPD is still not powerful enough. We suggest an extension to ACPD that allows us to solve the problem and illustrate with an example that this extension is very promising.

After the preliminaries, in Section 3, we introduce a fairly refined abstract domain, including type, mode and aliasing information, and we show, by means of an example, how ACPD allows us to analyze a coroutine and compile the transformed program. In Section 4, we formalize the operations used to analyze the example and prove correctness of the approach. In Section 5 we propose a more complex example and show why it is out of scope for ACPD. We introduce an additional abstraction in our domain and illustrate that this abstraction solves the problem. This abstraction, however, does not respect the requirements of the

formalization of ACPD in Leuschel [2004]. In Section 6, we discuss our prototype implementation. We end with a discussion.

A preliminary version of this paper appeared in De Schreye et al. [2014]. That version of the paper only provides the intuitions on how ACPD is able to provide a correct formalization for basic cases of CC. In the current paper we provide the full formal underpinning, including definitions and correctness results for Abstract unify, Abstract resolve, the order on the abstract domain, the widening operator and the abstraction function. We also prove the correctness of the approach, for the case in which the novel abstractions, introduced in Section 5, are not required. Also new in the current paper is that we further refine the abstract domain presented in De Schreye et al. [2014] and that we discuss our prototype implementation is Section 6.

2. Preliminaries

We assume that the reader is familiar with the basics of Logic Programming (Lloyd [1987]). We also assume knowledge of the basics of abstract interpretation (Bruynooghe [1991]) and of partial deduction (Lloyd and Shepherdson [1991]).

In this paper, names of variables will start with a capital. Names of constants will start with a lower case character. Given a program P, Con_P, Var_P, Fun_P and $Pred_P$ respectively denote the sets of all constants, variables, functors and predicate symbols in the language underlying P. $Term_P$ will denote the set of all terms constructable from Con_P, Var_P and Fun_P. $Atom_P$ denotes the set of all atoms which can be constructed from $Pred_P$ and $Term_P$. We will often need to refer to conjunctions of atoms of $Atom_P$ and we denote the set of all such conjunctions as $ConAtom_P$. Dom_P will denote $Term_P \cup Atom_P \cup ConAtom_P$.

We will introduce an abstract domain in the following section. The abstract domain will be based on a set of abstract variable symbols, $AVar_P$. Based on these, there is a corresponding set of abstract terms, $ATerm_P$, which consists of the terms that can be constructed from $AVar_P$ and Fun_P. For our purpose, abstract constants will not be required. $AAtom_P$ will denote the set of abstract atoms, being the atoms which can be constructed from $ATerm_P$ and $Pred_P$. $AConAtom_P$ denotes the set of conjunctions of elements of $AAtom_P$. Finally, $ADom_P$ will denote $ATerm_P \cup AAtom_P \cup AConAtom_P$.

3. An Example of a CC Transformation, Using ACPD

In this section, we provide the intuitions behind our approach by means of a simple example. First, we introduce the abstract domain. This domain consists of two types of variable symbols: g_i and $a_i, i \in \mathbb{N}_0$. The symbols g_i denote any ground term in the concrete language. The basic intuition for the symbols a_i is that they are intended to represent variables of the concrete domain. However, as we want the concretization of abstract terms to be closed under substitution (if an abstract term denotes some concrete term, then it should also denote all of its instances), an abstract variable a_i will actually represent any term of the concrete language.

The subscript i in a term g_i or a_i is used to represent aliasing. If an abstract term, abstract atom or abstract conjunction of atoms contains a_i several times (with the same subscript), the denoted concrete terms, atoms or conjunctions of atoms contain the *same* term in all positions corresponding to those occupied by a_i. Similarly, if a domain element contains g_i several times, the denoted concrete values contain the *same* ground term in all positions occupied by g_i. Note that if a_i and g_i share an index i, this has no particular meaning. As an example of the aliasing concept, the abstract conjunction $perm(g_1, a_1), ord(a_1)$ denotes the concrete conjunctions $\{perm(t_1, t_2), ord(t_2) \mid t_1, t_2 \in Term_P \text{ and } t_1 \text{ is ground}\}$.

Definition 1 (Abstract domain)
The abstract domain, $ADom_P$, is the union of:

- $AVar_P = \{g_i \mid i \in \mathbb{N}_0\} \cup \{a_i \mid i \in \mathbb{N}_0\}$.
- *$ATerm_P$, $AAtom_P$ and $AConAtom_P$, defined as the sets of the terms, atoms and conjunctions of atoms constructable from $AVar_P$, Fun_P and $Pred_P$.*

Next, we define the semantics of the abstract domain, through a concretization function γ. The definition

of γ will be based on functions that map abstract variables to concrete terms, called abstract-concrete substitutions.

Definition 2 (Abstract-concrete substitution)
An abstract-concrete substitution σ is a function from Dom_σ, a subset of $AVar_P$, into $Term_P$.

Note that, in Definition 2, we do not require that g_i symbols are mapped to ground terms. Of course, we will impose this extra requirement in the concretization function. However, we also need the more general notion of Definition 2, further in the paper.

Example 1 (Abstract-concrete substitution)
As an example, $\sigma = \{a_1/f(x), a_2/h(y, A), g_2/h(f(A), A)\}$.

For any $t \in ADom_P$, by $AVar(t)$ we denote the set of all abstract variables a_i and g_j occurring in t.

Definition 3 (Application of an abstract-concrete substitution)
Let $t \in ADom_P$ and σ an abstract-concrete substitution, such that $AVar(t) \subseteq Dom_\sigma$. The application of σ on t, denoted $t\sigma$, is an element of Dom_P obtained by replacing all occurrences of abstract variables a_i and g_j in t by their corresponding concrete term in σ.

Example 2
Let $t = perm(g_1, a_1), ord(a_1)$ and $\sigma = \{a_1/X, g_1/[s(0), s(s(0))]\}$. Then $t\sigma = perm([s(0), s(s(0))], X), ord(X)$.

Note that, due to the condition $AVar(t) \subseteq Dom_\sigma$, $t\sigma$ is concrete.

Definition 4 (Concretization function)
The concretization function $\gamma : ADom_P \rightarrow 2^{Dom_P}$ is defined as, for $t \in ADom_P$: $\gamma(t) = \{t\sigma \mid \sigma = \{a_{i_1}/t_1, \ldots, a_{i_n}/t_n\} \cup \{g_{j_1}/s_1, \ldots, g_{j_m}/s_m\}$ with σ an abstract-concrete substitution such that $AVar(t) \subseteq Dom_\sigma$ and for all $l = 1, \ldots, m : s_l$ is ground $\}$.

Example 3 (Concretization function)
$\gamma(p(f(a_2, g_1), a_1, a_2, q(h(a_1)))) = \{p(f(t_1, t_2), t_3, t_1, q(h(t_3))) \mid t_1, t_3 \in Term_P, t_2$ is a ground term of $Term_P\}$.

The abstract domain introduced above is infinitely large. There are two causes for this. Terms can be nested unboundedly deep, therefore infinitely many different terms exist. In addition, there are infinitely many a_i and $g_i, i \in \mathbb{N}_0$, symbols.

If so desired, the abstract domain can be refined, so that it becomes finite. This is done by using depth-k abstraction and by defining an equivalence relation on $\{a_i \mid i \in \mathbb{N}_0\}$ and on $\{g_i \mid i \in \mathbb{N}_0\}$. In Section 4, we define this equivalence relation. However, for the purpose of this paper, the infinite size of the abstract domain is not a problem.

We use permutation sort as an illustration of the concepts described above. Assume that instantiation-based delay declarations are used to interleave calls to *perm*/2 and *ord*/1 in the following program. The intention is to obtain a new program without delay declarations which interleaves calls to *perm*/2 and *ord*/1 in the same way.

Example 4 (Permutation sort)

```
sort(X,Y) ← perm(X,Y), ord(Y).
perm([],[]).
perm([X|Y],[U|V]) ←
   del(U,[X|Y],W),perm(W,V).
```

```
del(X,[X|Y],Y).
del(X,[Y|U],[Y|V]) ← del(X,U,V).
ord([]).
ord([X]).
ord([X,Y|Z]) ← X ≤ Y, ord([Y|Z]).
```

The SWI-Prolog code in Listing 1 interleaves the calls in such a way, but it uses the non-logical *when/2* predicate and requires atoms to be reordered. It uses a predicate *permsort/2* rather than *sort/2* to avoid a name clash with SWI-Prolog's built-in.

We now show how ACPD can be used to "compile" the control. ACPD requires a top-level abstract atom (or conjunction) to start the transformation. Let $sort(g_1, a_1)$ be this atom.

Central to the transformation is the $\mathcal{A}$-coveredness condition of partial deduction. Stated somewhat informally and within the context of *abstract conjunctive* partial deduction, $\mathcal{A}$-coveredness means that, given a finite set of finite abstract deduction trees, obtained for queries from a set $\mathcal{A}$ of abstract conjunctions, the conjunctions in the leaves of all these trees can be split into new abstract conjunctions, $\wedge_{i=1,\ldots,n} C_i$, such that, for every C_i, there exists an $A_i \in \mathcal{A}$ with $\gamma(C_i) \subseteq \gamma(A_i)$.

```
permsort(X,Y) :-
  when(;(ground(Y),?=(Y,[_A])),ord(Y)), perm(X,Y).

perm([],[]).
perm([X|Y],[U|V]) :- select(U,[X|Y],W), perm(W,V).

ord([]).
ord([_]).
ord([X,Y|Z]) :-
  freeze(Y, X =< Y),
  when(;(ground([Y|Z]),?=([Y|Z],[_A])),ord([Y|Z])).
```

Listing 1: SWI-Prolog implementation of a coroutining permutation sort

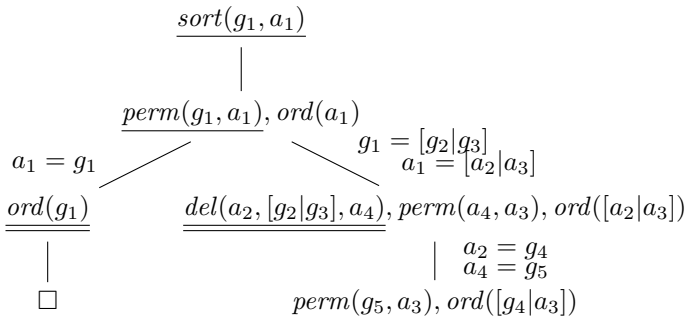

Figure 1. Abstract tree for $sort(g_1, a_1)$

Here, our initial set $\mathcal{A}$ is $\{sort(g_1, a_1)\}$. Below, we construct a finite number of finite, abstract partial deduction derivation trees for abstract (conjunctions of) atoms. The construction of these trees assumes an "abstract unification" and an "abstract resolve" operation. Their formal definitions can be found in Section 4. For now, we only show their effects in abstract partial derivation trees.

Next, we need an "oracle" that decides on the selection rule applied in the abstract derivation trees. This oracle mainly has two functions:

- to decide whether an obtained goal should be unfolded further, or whether it should be kept residual (to be split and added to $\mathcal{A}$),
- to decide which atom of the current goal should be selected for resolving.

In fact, we will use a third type of decision that the oracle may make: it may decide to "fully evaluate" a selected atom. This type of decision is not commonly supported in partial deduction. What it means is that we decide not to transform a certain predicate of the original program, but merely keep its original definition in the transformed program. In partial deduction, this can be done by never selecting these atoms, including them in $\mathcal{A}$ and including their original definition in the transformed program.

In our setting, however, we want to know the effect that solving the atom has on the remainder of the goal. Therefore, we will assume that a full abstract interpretation over our abstract domain computes the abstract bindings that solving the atom results in. These are applied to the remainder of the goal. Note that this cannot easily be done in standard partial deduction, as fully evaluating an atom during (concrete) partial deduction may not terminate. In Vidal [2011], a similar functionality is integrated in a hybrid approach to conjunctive partial deduction.

We will not specify the exact mechanism for the oracle here. In our implementation of the technique (see Section 6), the atom selection aspect of the oracle is implemented using instantiation-based delay statements which are orthogonal to the program definition.

Figures 1, 2 and 3 show the abstract partial derivation trees that ACPD builds for permutation sort and top level $\mathcal{A} = \{sort(g_1, a_1)\}$.

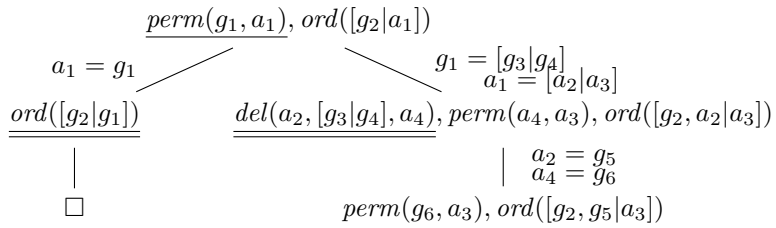

Figure 2. Abstract tree for $perm(g_1, a_1), ord([g_2|a_1])$

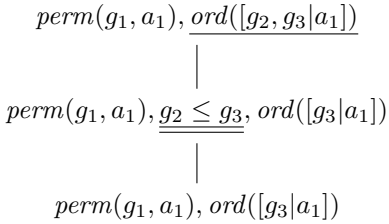

Figure 3. Abstract tree for $perm(g_1, a_1), ord([g_2, g_3|a_1])$

In these figures, in each goal, the atom selected for abstract resolution is underlined. If an atom is underlined twice, this expresses that the atom was selected for full abstract interpretation.

Both resolving and full abstract evaluation may create abstract bindings. Our abstract unification collects abstract bindings made on the a_i and g_i variables.

A goal with no underlined atom indicates that the oracle selects no atom and decides to keep the conjunction residual. After the construction of the tree in Figure 1, ACPD adds the abstract conjunction $perm(g_5, a_3), ord([g_4|a_3])$ to $\mathcal{A}$, as part of its aim to eventually reach an $\mathcal{A}$-covered set $\mathcal{A}$. ACPD starts a new tree for this atom. A renaming of this tree is shown in Figure 2.

This tree is quite similar to the subtree rooted at the second level of the tree in Figure 1. The main difference is that, in the rightmost residual leaf, the *ord* atom now has a list argument with two g_i elements. This pattern does not yet exist in the current $\mathcal{A}$ and is therefore added to $\mathcal{A}$. A third abstract tree is computed for a renaming of $perm(g_4, a_3), ord([g_1, g_3|a_3])$, shown in Figure 3.

In Figure 3, the residual leaf $perm(g_1, a_1), ord([g_3|a_1])$ is a renaming of a conjunction which is already contained in $\mathcal{A}$.

To avoid confusion in the notation for a set of (abstract) conjunctions, we will denote a conjunction of (abstract) atoms, $A_1, \ldots, A_n$, as $\wedge(A_1, \ldots, A_n)$, where we assume the existence of $\wedge/n$ functors for all possible arities n.

Returning to the example, ACPD now concludes that $\mathcal{A}$-coveredness holds for $\mathcal{A} = \{sort(g_1, a_1), \wedge(perm(g_3, a_3), ord([g_2|a_3])), \wedge(perm(g_4, a_3), ord([g_1, g_3|a_3]))\}$. In standard (concrete) conjunctive partial deduction, the analysis phase would now be completed. In ACPD, however, we need an additional step. In the abstract partial derivation trees, we have not collected the concrete bindings that unfolding would produce. These are required to generate the resolvents. Therefore, we need an additional step, constructing essentially the same three trees again, but now using concrete terms and concrete unification.

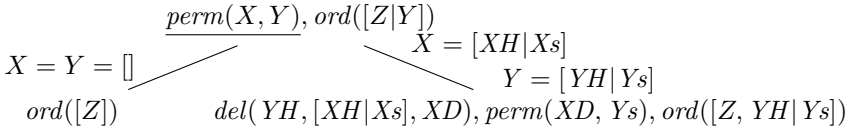

Figure 4. Concrete tree corresponding to Figure 2

We only show one of these concrete derivation trees in Figure 4. It corresponds to the tree in Figure 2. We define the root of a concrete derivation tree corresponding to an abstract tree as follows.

Definition 5 (Concrete conjunctions in the root)
Let $acon \in \mathcal{A}$. Let $\sigma = \{a_{i_1}/t_1, \ldots, a_{i_n}/t_n\} \cup \{g_{j_1}/s_1, \ldots, g_{j_m}/s_m\}$ be an abstract-concrete substitution with $AVar(acon) \subseteq Dom_\sigma$, such that t_k and s_l, for $k = 1, \ldots, n$ and $l = 1, \ldots, n$, are distinct variables of Var_P.
The conjunction in the root of the corresponding concrete tree for acon, denoted $c(acon)$, is acon σ.

Note that $c(acon)$ is uniquely defined, up to variable renaming.

When unfolding the concrete tree, every abstract resolution of the abstract tree is mimicked, using the same clauses, over the concrete domain. For some of these unfoldings, the concrete unification may fail: the abstract resolution is only a safe approximation of the concrete one. As a result, some branches of the abstract tree may be aborted in the concrete one.

The step of full abstract interpretation of the $ord(g_1)$ and $del(a_2, [g_3|g_4]), a_4)$ atoms in Figure 2 has no counterpart in Figure 4. The atoms $ord([L])$ and $del(YSH, [MH|MS], YSN)$ are kept residual and the $ord/1$ and $del/3$ clauses are added to the transformed program.

As a next step, we derive resultants from the concrete counterparts to the abstract trees. We obtain the following resultants from the tree in Figure 4:

$perm([], []), ord([Z]) \leftarrow ord([Z])$.

$perm([XH|Xs], [YH|Ys]), ord([Z, YH|Ys]) \leftarrow del(YH, [XH|Xs], XD), perm(XD, Ys), ord([Z, YH|Ys])$.

From the concrete counterpart of the tree in Figure 1, we get:

$sort([], []) \leftarrow ord([])$.

$sort([XH|Xs], [YH|YS]) \leftarrow del(YH, [XH|Xs], XD), perm(XD, Ys), ord([YH|Ys])$.

From the concrete counterpart of the tree in Figure 3, we get:

$perm(W, X), ord([Y, Z|X]) \leftarrow Y \leq Z, perm(W, X), ord([Z|X])$.

Wrapping up the transformation, we rename and filter $\wedge(perm(X, Y), ord([Z|Y]))$ to $p_1(X, Y, Z)$ and rename and filter $\wedge(perm(W, X), ord([Y, Z|X]))$ to $p_2(W, X, Y, Z)$.

Renaming $sort/2$ to $permsort/2$ (to avoid a name clash with SWI-Prolog's built-in), this translates directly to the Prolog program in Listing 2:

```
del(X,[X|Y],Y).
del(X,[Y|U],[Y|V]) :- del(X,U,V).

ord([]).
ord([_]).
ord([X,Y|Z]) :- X =< Y, ord([Y|Z]).

p1([],[],Z) :- ord([Z]).
p1([XH|Xs],[YH|Ys],Z) :- del(YH,[XH|Xs],XD), p2(XD,Ys,Z,YH).
permsort([],[]) :- ord([]).
permsort([XH|Xs],[YH|Ys]) :- del(YH,[XH|Xs],XD), p1(XD,Ys,YH).
p2(W,X,Y,Z) :- Y =< Z, p1(W,X,Z).
```

Listing 2: Synthesized implementation of coroutining permutation sort

This program achieves our initial aim. When executed under the standard left-to-right selection rule, it interleaves the stepwise generation of a permutation with the check for ordering. As a result, the generation of a permutation is aborted as soon as it contains two elements which are out of order.

4. A More Formal Account

With the exception of the formal definition of the abstract domain and the concretization function, Section 3 only provides the intuitions of the approach, based on an example. In the current section, we formally present the key concepts. This includes the central operations of abstract unification and abstract resolve.

Abstract resolve, in turn, requires the ability to generate abstract versions of the program clauses. We introduce an order on the abstract domain and an abstraction function, α, which allows us to compute these abstract clauses.

Then, we discuss correctness of the transformed programs. We show that the preconditions for the correctness results of the ACPD approach are fulfilled, so that the main correctness theorem of Leuschel [2004] applies.

Finally, we briefly discuss control issues and widening.

4.1. Abstract Unification

We adapt the standard unification algorithm of Martelli and Montanari [1982] to $ADom_P$. In Algorithm 1, both the symbols a_i and g_i take the role of the variables in the Martelli-Montanari algorithm. For the a_i symbols, this seems obvious, as a_i represents any concrete term (including a variable). For the g_i symbols, as g_i represents any ground term, the abstract unification algorithm needs to be opportunistic in their treatment in the unification. The abstract unification $g_i = t$ opportunistically always succeeds, assuming that g_i represents a ground instance of t.

As a result of the use of abstract variables, the majority of the rewrite steps in Algorithm 1 are the direct counterparts of the rewrites in the Martelli-Montanari algorithm, reformulated both for the a_i's and the g_i's.

In addition, we need to deal with the fact that our variables are colored: with color either "a" or "g". The "g" color is dominant. Any "a"-colored variable becomes "g"-colored, after unification with a "g"-colored variable. This requires two additional rewrite steps with respect to the Martelli-Montanari algorithm.

Algorithm 1 (Abstract unification)
Let $s1, s2 \in ATerm_P \cup AAtom_P$. Let $mgu = \{s1 = s2\}$. Repeatedly select an equality e and perform any of the following transformations on mgu. If no transformation applies, stop with success.

Step 1: remove any equality of the form $t = t$

Step 2: select any equality $t = s$, with s of the form a_i or g_i, with t not of the form a_j or g_j and replace it by $s = t$

Step 3: select any equality $s = t$, with s of the form a_i or g_i, where t contains s as a subterm and stop with failure

Step 4: select any equality $g_i = a_j$, replace it by $a_j = g_i$ and replace all other occurrences of a_j in mgu by g_i

Step 5: select any equality $g_i = t$, where t contains $a_{i_1}, \ldots, a_{i_n}$, with $n > 0$, and where t is not an a_j and do the following:

- introduce $g_{k_1}, \ldots, g_{k_n}$, with $k_1, \ldots, k_n$ fresh indices from $\mathbb{N}_0$
- replace all occurrences of a_{i_j} in mgu by g_{k_j}, for all $j = 1, \ldots, n$
- add equalities $a_{i_j} = g_{k_j}$, for all $j = 1, \ldots, n$, to mgu

Step 6: select any equality $s = t$, with s of the form a_i or g_i, where s occurs elsewhere in mgu and replace all other occurrences of s in mgu by t

Step 7: select any equality of the type $f(t_1, \ldots, t_n) = g(s_1, \ldots, s_m)$ and stop with failure if $f \neq g$ or $n \neq m$; replace $f(t_1, \ldots, t_n) = g(s_1, \ldots, s_m)$ in mgu by $t_1 = s_1, \ldots, t_n = s_n$ otherwise

In Algorithm 1, five out of seven transformations for mgu are the direct counterparts of the rewrites in the Martelli-Montanari algorithm. They only differ from the concrete versions in that a_i and g_i symbols take on

the role of the variables. The two additional rewrites, steps 4 and 5, deal with the "g"-coloring of a_i symbols. Both these rewrites make sure that, if an a_i symbol is unified with a g_j symbol, it is replaced by a (possibly fresh) g_k symbol. Due to the similarity with the Martelli-Montanari algorithm, the proofs for termination and correctness are easily adapted from Martelli and Montanari [1982].

Proposition 1 (Termination of Abstract unify)
Algorithm 1 terminates.

Proof. Similar to the proof in Martelli and Montanari [1982], we introduce the following numbers:

- n_a is the number of a_i symbols in *mgu* that occur more than once **or** do not occur as the left-hand side of an equality
- n_g is the number of g_i symbols in *mgu* that occur more than once **or** do not occur as the left-hand side of an equality
- n_{fun} is the total number of occurrences of function symbols in *mgu*
- n_{equ} is the number of equalities in *mgu* of the type $s = s$ or $t = a_i$, with t not of the form a_j or g_j, or $t = g_i$, with t not of the form a_j or g_j

It suffices to prove that the lexicographic order on 4-tuples $(n_a, n_g, n_{fun}, n_{equ})$ strictly decreases with every application of a rewrite on *mgu*. As the lexicographic order is well-founded, there can only be a finite number of consecutive strict decreases.

We will not discuss our counterparts of the five rewrites in the concrete unification algorithm. The arguments for these rewrites are completely similar to those in Martelli and Montanari [1982].

For step 4, n_a decreases by one. Therefore $(n_a, n_g, n_{fun}, n_{equ})$ decreases. For step 5, the term t contains at least one a_i. Therefore, again, n_a decreases by at least one and the lexicographic order decreases.

Note that these two steps are the reason why we need two separate variables n_a and n_g in the order, instead of just one for the union of all a_i and g_i (as it is the case in Martelli and Montanari [1982]). For step 5, n_g increases with the number of fresh variables that are introduced. Also step 4 may increase n_g. □

To show the correctness of Algorithm 1, we need the ability to "apply" the outcome of Abstract unify to abstract atoms and terms. We need the following definitions and properties.

Definition 6 (Solved form)
A set AE of equalities of elements of $ATerm_P$ is in solved form if it is of the form $\{a_{i_1} = t_1, \ldots, a_{i_n} = t_n, g_{j_1} = s_1, \ldots, g_{j_m} = s_m\}$, where:

- *all $i_k, k = 1, \ldots, n$ are distinct and all $j_l, l = 1, \ldots, m$, are distinct*
- *no a_{i_k} occurs in $t_1, \ldots, t_n$, nor in $s_1, \ldots, s_m$*
- *no g_{j_l} occurs in $t_1, \ldots, t_n$, nor in $s_1, \ldots, s_m$*
- *no a_i variable occurs in $s_1, \ldots, s_m$*

Due to the conditions above, we can associate an abstract substitution with any set of equalities AE in solved form.

Definition 7 (Abstract substitution)
An abstract substitution Θ_A is a finite set of ordered pairs in $AVar_P \times ATerm_P$, such that there exists a set AE of equalities of elements of $ATerm_P$ in solved form, with $\Theta_A = \{x/y \mid x = y \in AE\}$.

Using, again, the conditions in Definition 6, it is clear that abstract substitutions can be applied to elements of $ADom_P$.

Definition 8 (Application of an abstract substitution)
Let $s \in ADom_P$ and let Θ_A be an abstract substitution. Θ_A can be applied to s, resulting in $s\Theta_A \in ADom_P$, obtained by replacing all a_i and g_j in s occurring on the left-hand side of a pair in Θ_A, by the corresponding right-hand side.

Note that we can also apply an abstract substitution Θ_A to a set of equalities AE between elements of $ATerm_P$. This is done by applying the abstract substitution to every term in AE. The result is denoted $AE\Theta_A$.

We can now define an abstract unifier.

Definition 9 (Abstract unifier)
Let $s_1, s_2 \in ADom_P$. An abstract substitution, Θ_A, is an abstract unifier for (s_1, s_2), if $s_1\Theta_A = s_2\Theta_A$.

Let $AE = \{s_{i_1} = s_{i_2} \mid i = 1, \ldots, n\}$ be a set of equalities between elements of $ATerm_P$. An abstract substitution Θ_A is an abstract unifier for AE if Θ_A is an abstract unifier for every pair (s_{i_1}, s_{i_2}).

Observe that an abstract substitution Θ_A is an abstract unifier for AE if and only if $AE\Theta_A$ is a set of syntactic identity equalities.

In order to prove that the final set *mgu* computed by Algorithm 1 corresponds to an abstract unifier for the initial abstract terms or atoms s_1 and s_2, we still need the following proposition and definitions.

Proposition 2 (Solved form in Algorithm 1)
If Algorithm 1 terminates with success, then the final set mgu is in solved form.

Proof. It is easy to check that, in case any of the conditions of Definition 6 is not met, one of the rewrites of Algorithm 1 is still applicable. This contradicts successful termination of the algorithm. □

As a result of Proposition 2, we can associate with the final set *mgu* produced by Algorithm 1, in the case that the algorithm terminates successfully, an abstract substitution which we will denote as Θ_{mgu}. If we want to be explicit concerning the $s_1, s_2 \in ADom_P$ on which we applied the algorithm, we denote it as $\Theta_{mgu}(s_1, s_2)$.

Proposition 3 (Correctness of Abstract unify (1))
Let $s_1, s_2 \in ATerm_P \cup AAtom_P$ and let $\Theta_{mgu}(s_1, s_2)$ be the abstract substitution associated with the final set mgu obtained by successfully terminating Algorithm 1 on (s_1, s_2). Then, Θ_{mgu} is an abstract unifier for (s_1, s_2).

Proof. The proof is similar to the one in Martelli and Montanari [1982]. First note that for any set of equalities in solved form AE, the associated abstract substitution Θ_A is an abstract unifier for AE. This is easily verified: the application of Θ_A on the equalities of AE reduces them to syntactic identity equalities. Thus, Θ_{mgu} is an abstract unifier for the last set *mgu* in Algorithm 1, by Proposition 2.

It now suffices to prove that every abstract unifier of the *mgu* set obtained after any rewrite step of Algorithm 1, is also an abstract unifier of the *mgu* set before that rewrite step.

We do not discuss the argument for the five rewrite rules which are direct counterparts of the rules in Martelli and Montanari [1982]. The argument is identical to that in Martelli and Montanari [1982].

Consider step 4 in Algorithm 1. Let $AE \cup \{g_j = a_i\}$ be the set *mgu* before step 4. Let $\Theta = \{a_i/g_j\}$. Then the next set *mgu* is $AE\Theta \cup \{a_i = g_j\}$.

Let Θ_A be any abstract unifier for $AE\Theta \cup \{a_i = g_j\}$. Then, $AE\Theta\Theta_A$ consists of syntactic identity equalities and $a_i\Theta_A = g_j\Theta_A$ is a syntactic identity equality. Now consider $AE\Theta_A$. AE only differs from $AE\Theta$ in that some occurrences of g_j of $AE\Theta$ are replaced by occurrences of a_i in AE. But, as $a_i\Theta_A$ and $g_j\Theta_A$ are syntactically equal, $AE\Theta_A = AE\Theta\Theta_A$ and therefore it only consists of syntactic identity equalities. Thus, Θ_A is a unifier for $AE \cup \{g_j = a_i\}$.

Next, consider step 5 in Algorithm 1. Let $AE \cup \{g_i = t\}$, where t contains $a_{i_1}, \ldots, a_{i_n}, n > 0$, be the set *mgu* before step 5. Let $\Theta = \{a_{i_1}/g_{k_1}, \ldots, a_{i_n}/g_{k_n}\}$, with $k_1, \ldots, k_n$ fresh indices from $\mathbb{N}_0$. Step 5 transforms *mgu* into $AE' = AE\Theta \cup \{g_i = t\}\Theta \cup \{a_{i_1} = g_{k_1}, \ldots, a_{i_n} = g_{k_n}\}$.

Let Θ_A be an abstract unifier for AE'. Then, $AE\Theta\Theta_A$ consists of syntactic identity equalities and $g_i\Theta\Theta_A = t\Theta\Theta_A$, $a_{i_1}\Theta_A = g_{k_1}\Theta_A, \ldots, a_{i_n}\Theta_A = g_{k_n}\Theta_A$ are syntactic identity equalities. $AE \cup \{g_i = t\}$ only differs from $AE\Theta \cup \{g_i = t\}\Theta$ in that all occurrences of $g_{k_1}, \ldots, g_{k_n}$ in $AE\Theta \cup \{g_i = t\}\Theta$ are replaced by corresponding $a_{i_1}, \ldots, a_{i_n}$ in $AE \cup \{g_i = t\}$. Thus, as $a_{i_1}\Theta_A = g_{k_1}\Theta_A, \ldots, a_{i_n}\Theta_A = g_{k_n}\Theta_A$ are identities, $AE\Theta_A \cup \{g_i = t\}\Theta_A = AE\Theta\Theta_A \cup \{g_i = t\}\Theta\Theta_A$ and therefore it consists only of syntactic identity equalities. Thus, Θ_A is an abstract unifier for $AE \cup \{g_i = t\}$. □

Note that there is a second correctness concern related to Abstract unify. Namely, the fact that the abstract unification correctly mimics the concrete unification. In order to prove this second correctness result, we need the following concept and lemma.

Definition 10 (*a-g*-extension)
Let σ be an abstract-concrete substitution. An abstract-concrete substitution σ' is an a-g-extension of σ, denoted $\sigma' \succcurlyeq \sigma$, if there exists a set $\{a_{i_1}, \ldots, a_{i_n}\} \subseteq Dom_\sigma$ and a set $\{g_{k_1}, \ldots, g_{k_n}\}$ of abstract variables not occurring in Dom_σ, such that:

1. $Dom_{\sigma'} = Dom_\sigma \cup \{g_{k_1}, \ldots, g_{k_n}\}$
2. $\sigma'(t) = \sigma(t)$, *for* $t \in Dom_\sigma$
3. $\sigma'(g_{k_j}) = \sigma(a_{i_j})$, *for* $j = 1, \ldots, n$

Example 5
Let $\sigma = \{g_1/f(X,Y), a_1/X, a_2/Y\}$. Then, $\sigma' = \{g_1/f(X,Y), a_1/X, a_2/Y, g_2/X, g_3/Y\}$ is an *a-g*-extension of σ.

Lemma 1 (Transitivity of $\succcurlyeq$)
The relation $\succcurlyeq$ is transitive.

Proof. Let $\sigma'' \succcurlyeq \sigma'$ and let $\sigma' \succcurlyeq \sigma$. In order to prove transitivity of $\succcurlyeq$, we need to establish that conditions 1, 2 and 3 are maintained between σ'' and σ. For condition 1, we have $Dom_{\sigma''} = Dom_{\sigma'} \cup \{g_{l_1}, \ldots, g_{l_m}\}$ by definition. As $Dom_{\sigma'} = Dom_\sigma \cup \{g_{k_1}, \ldots, g_{k_n}\}$, condition 1 is indeed maintained.

For condition 2, $\sigma''(t) = \sigma'(t)$, for $t \in Dom_{\sigma'}$ by definition. As $\sigma'(t) = \sigma(t)$, for $t \in Dom_\sigma$, condition 2 is similarly maintained.

For condition 3, first consider the variables $\{g_{l_1}, \ldots, g_{l_m}\}$ introduced in the *a-g*-extension $\sigma'' o f \sigma'$. We have $\sigma''(g_{l_j}) = \sigma'(a_{i_j})$, for $j = 1, \ldots, m$, for some $a_{i_j} \in Dom_{\sigma'}$. By condition 1, if $a_{i_j} \in Dom_{\sigma'}$, then $a_{i_j} \in Dom_\sigma$. By condition 2, $\sigma'(a_{i_j}) = \sigma(a_{i_j})$. Thus, $\sigma''(g_{l_j}) = \sigma(a_{i_j})$. For the variables $\{g_{k_1}, \ldots, g_{k_n}\}$ introduced in the *a-g*-extension σ' of σ, $\sigma''(g_{k_j}) = \sigma'(g_{k_j})$, because condition 2 holds for σ'' and σ'. The result then follows from condition 3 on $\sigma' \succcurlyeq \sigma$. □

Example 6
Let $s_1 = g_1$ and let $s_2 = f(a_1, a_2)$. Then, an abstract-concrete substitution σ which makes $(s_1 = s_2)$ a syntactic equality is $\sigma = \{g_1/f(X,Y), a_1/X, a_2/Y\}$. A set *mgu* for s_1 and s_2 is $mgu = \{g_1 = f(g_2, g_3), a_1 = g_2, a_2 = g_3\}$. Applying σ to *mgu* does not provide us with a set of syntactic equalities.

Let $\sigma' = \{g_1/f(X,Y), a_1/X, a_2/Y, g_2/X, g_3/Y\}$. Applying σ' to *mgu* does provide a set of syntactic equalities. Moreover, σ' is an *a-g*-extension of σ.

For the following lemma, we are assuming that Algorithm 1 successfully terminates, given a starting equality $s_1 = s_2, s_1, s_2 \in ATerm_P \cup AAtom_P$. With $mgu(s_1, s_2)$, we denote the final *mgu* set. We also denote by $AVar(s_1, s_2)$ the abstract variables occurring in s_1 or s_2.

Lemma 2
Let σ be an abstract-concrete substitution, with $Dom_\sigma = AVar(s_1, s_2)$, such that $\sigma(s_1 = s_2)$ is a syntactic identity equality and such that, for any $g_i \in Dom_\sigma$, $\sigma(g_i)$ is a ground term. Then, there exists an a-g-extension σ' of σ, such that $\sigma'(mgu(s_1, s_2))$ is a set of syntactic identity equalities and such that, for any $g_i \in Dom_{\sigma'}$, $\sigma'(g_i)$ is a ground term.

Proof. Consider any rewrite step in Algorithm 1. We denote the set *mgu* before the step as mgu_1 and the set *mgu* after the step as mgu_2. We will prove that, if σ is an abstract-concrete substitution such that $\sigma(mgu_1)$ is a set of syntactic identity equalities and such that, for any $g_i \in Dom_\sigma$, $\sigma(g_i)$ is a ground term, then there exists an *a-g*-extension σ' of σ, such that $\sigma'(mgu_2)$ is a set of syntactic identity equalities and such that, for any $g_i \in Dom_{\sigma'}$, $\sigma'(g_i)$ is a ground term.

As the relation $\succcurlyeq$ is transitive by Lemma 1, the lemma will follow directly from this proof.

So, consider step 1 in Algorithm 1. Here, $mgu_2 = mgu_1 \backslash \{t = t\}$. Clearly, with $\sigma' = \sigma$, if $\sigma(mgu_1)$ is a set of syntactic identity equalities, then so is $\sigma'(mgu_2)$.

For step 2, mgu_1 contains either $t = a_i$ or $t = g_i$, while in mgu_2, this equality is replaced by $a_i = t$, respectively $g_i = t$. Obviously, with $\sigma' = \sigma$, the result holds.

Step 3 does not need to be considered, as we are assuming that Algorithm 1 successfully terminates.

For step 4, mgu_1 contains $g_i = a_j$ and mgu_2 is obtained from mgu_1 by replacing $g_i = a_j$ by $a_j = g_i$ and by replacing all other occurrences of a_j in mgu_1 by g_i. Again, let $\sigma' = \sigma$. As $\sigma(g_i = a_j)$ is a syntactic identity equality, so is $\sigma'(a_j = g_i)$. Moreover, all other equalities in mgu_2 only differ from those in mgu_1 by replacements of a_j by g_i. As $\sigma(a_j) = \sigma'(g_i)$, $\sigma(mgu_1)$ being syntactic identity equalities implies the same for $\sigma'(mgu_2)$.

Only step 5 requires the notion of an *a-g*-extension. Here, mgu_1 contains an equality $g_i = t$, where t contains $a_{i_1}, \ldots, a_{i_n}$, with $n > 0$. The step introduces new variables $g_{k_1}, \ldots, g_{k_n}$ and mgu_2 is obtained from mgu_1 by replacing all occurrences of a_{i_j} by $g_{k_j}, j = 1, \ldots, n$, and adding the equalities $a_{i_j} = g_{k_j}, j = 1, \ldots, n$.

Let σ' be the *a-g*-extension of σ based on the sets $\{a_{i_1}, \ldots, a_{i_n}\}$ and $\{g_{k_1}, \ldots, g_{k_n}\}$. By Definition 10:

$\sigma'(g_{k_j}) = \sigma(a_{i_j})$ and $\sigma'(a_{i_j}) = \sigma(a_{i_j})$, for $j = 1, \ldots, n$. Thus, $\sigma'(a_{i_j} = g_{k_j})$ are syntactic identity equalities, for $j = 1, \ldots, n$. As all other equalities in mgu_2 are obtained by replacing a_{i_j}'s by g_{k_j}'s in the equalities in mgu_1, $\sigma(mgu_1)$ are syntactic identity equalities and $\sigma'(g_{k_j})$'s are identical to $\sigma(a_{i_j})$'s, $\sigma'(mgu_2)$ is also a set of syntactic identity equalities.

For this step, we have the additional proof obligation on the groundness of σ' on newly introduced variables g_{k_j}. It is given that mgu_1 contains the equality $g_i = t$ and that $\sigma(g_i)$ is ground. Thus, $\sigma(t)$ is ground. As $a_{i_1}, \ldots, a_{i_n}$ are subterms of t, $\sigma(a_{i_1}), \ldots, \sigma(a_{i_n})$ are ground. Therefore, by $\sigma'(g_{k_j}) = \sigma(a_{i_j})$, $j = 1, \ldots, n$, $\sigma'(g_{k_j})$ is ground.

For step 6, mgu_1 contains an equality $a_i = t$ or $g_i = t$. Here, mgu_2 is obtained from mgu_1 by replacing all other occurrences of a_i, respectively g_i, by t. Let $\sigma' = \sigma$. As $\sigma(a_i = t)$, respectively $\sigma(g_i = t)$, is a syntactic identity equality, $\sigma'(mgu_2) = \sigma(mgu_1)$, so the result holds.

Finally, for step 7, mgu_1 contains an equality $f(t_1, \ldots, t_n) = g(s_1, \ldots, s_m)$. As we assume that Algorithm 1 terminates successfully, $f = g$, $n = m$ and mgu_2 is obtained from mgu_1 by replacing the equality above by $t_1 = s_1, \ldots, t_n = s_n$.

Let $\sigma' = \sigma$. As $\sigma(f(t_1, \ldots, t_n) = g(s_1, \ldots, s_n))$ is a syntactic identity equality, so are $\sigma'(t_1 = s_1), \ldots, \sigma'(t_n = s_n)$. All other equalities of $\sigma'(mgu_2)$ are in $\sigma(mgu_1)$, so the result holds. □

Proposition 4 (Correctness of Abstract unify (2))
Let $s_1, s_2 \in ATerm_P \cup AAtom_P$, such that $AVar(s_1) \cap AVar(s_2) = \emptyset$ and let $\Theta_{mgu}(s_1, s_2)$ be the abstract unifier associated with the final set mgu obtained by successfully terminating Algorithm 1 on (s_1, s_2). For any $cs_1 \in \gamma(s_1)$ and $cs_2 \in \gamma(s_2)$, if cs_1 and cs_2 unify with unifier δ_{mgu}, then $cs_1\delta_{mgu} = cs_2\delta_{mgu} \in \gamma(s_1\Theta_{mgu}(s_1, s_2))$.

Proof. As $cs_1 \in \gamma(s_1)$ and $cs_2 \in \gamma(s_2)$, there exist abstract-concrete substitutions σ_1 and σ_2, such that $cs_1 = \sigma_1(s_1)$, $cs_2 = \sigma_2(s_2)$ and such that for all $g_{i_j} \in AVar(s_1)$, $\sigma_1(g_{i_j})$ is ground and for all $g_{k_j} \in AVar(s_2)$, $\sigma_2(g_{i_j})$ is ground.

As δ_{mgu} exists, $cs_1\delta_{mgu} = cs_2\delta_{mgu} = \sigma_1(s_1)\delta_{mgu} = \sigma_2(s_2)\delta_{mgu}$. Considering δ_{mgu} as a function, we define ψ_1 to be the composition of σ_1 and δ_{mgu} and ψ_2 to be the composition of σ_2 and δ_{mgu}. Thus, we have:

$$cs_1\delta_{mgu} = cs_2\delta_{mgu} = \psi_1(s_1) = \psi_2(s_2) \tag{1}$$

As $AVar(s_1) \cap AVar(s_2) = \emptyset$, next we define an abstract-concrete substitution σ on $AVar(s_1) \cup AVar(s_2)$:

$\sigma(t) = \psi_1(t)$, for $t \in AVar(s_1)$

$\sigma(t) = \psi_2(t)$, for $t \in AVar(s_2)$

From Equation (1), we get:

$$cs_1\delta_{mgu} = cs_2\delta_{mgu} = \sigma(s_1) = \sigma(s_2) \tag{2}$$

As $\sigma(s_1) = \sigma(s_2)$, $\sigma(s_1 = s_2)$ is a syntactic identity equality. Moreover, as $\sigma_1(g_{i_j})$ is ground on all $g_{i_j} \in AVar(s_1)$ and $\sigma_2(g_{k_j})$ is ground on all $g_{k_j} \in AVar(s_2)$, so is σ. Thus, by Lemma 2 there exists an a-g-extension σ' of σ, such that $\sigma'(mgu(s_1, s_2))$ is a set of syntactic identity equalities and such that, for any $g_i \in Dom_{\sigma'}, \sigma'(g_i)$ is ground.

Now, consider $\sigma'(s_1\Theta_{mgu(s_1,s_2)})$. As $\sigma'(mgu(s_1, s_2))$ is a set of syntactic identity equalities, for every pair s/t in $mgu(s_1, s_2)$, $\sigma'(s)$ is identical to $\sigma'(t)$. As $\Theta_{mgu(s_1,s_2)}$ only replaces such s's by t's in s_1, $\sigma'(s_1\Theta_{mgu(s_1,s_2)}) = \sigma'(s_1)$.

By definition, an a-g-extension σ' of an abstract-concrete substitution σ satisfies $\sigma'|_{Dom_\sigma} = \sigma$. Thus, $\sigma'(s_1\Theta_{mgu(s_1,s_2)}) = \sigma(s_1)$.

Then, from Equation (2), we obtain $cs_1\delta_{mgu} = cs_2\delta_{mgu} = \sigma'(s_1\Theta_{mgu(s_1,s_2)})$.

As σ' is ground for any $g_i \in Dom_\sigma$, we can conclude that $cs_1\delta_{mgu} = cs_2\delta_{mgu} \in \gamma(s_1\Theta_{mgu(s_1,s_2)})$. □

Note that the condition that $AVar(s_1) \cap AVar(s_2) = \emptyset$ is necessary for the proposition to hold. This is shown in the following example.

Example 7
Let $s_1 = f(a_1, g_1)$ and $s_2 = f(g_1, a_1)$. Algorithm 1 terminates successfully for $s_1 = s_2$ and $mgu(s_1, s_2) = \{a_1/g_1\}$. We have that $s_1\Theta_{mgu(s_1,s_2)} = s_2\Theta_{mgu(s_1,s_2)} = f(g_1, g_1)$.

Take $cs_1 = f(X, p)$ and $cs_2 = f(q, Y)$. Clearly, $cs_1 \in \gamma(s_1)$ and $cs_2 \in \gamma(s_2)$. Also, cs_1 and cs_2 unify, with $\delta_{mgu} = \{X/q, Y/p\}$ and resulting term $cs_1\delta_{mgu} = f(p, q)$. However, $f(p, q) \notin \gamma(f(g_1, g_1))$.

4.2. Abstract resolve

Next, we need a more formal account of the Abstract resolve operation. To do this, it is useful to have a definition of the abstraction function, so that we can map concrete clauses to their abstract counterparts. To define the abstraction function, in turn, it is useful to first introduce an order relation on $ADom_P$.

Definition 11 (Pre-ordering on $ADom_P$)
Let $t_1, t_2 \in ADom_P$.
$t_1 \succcurlyeq t_2$ if and only if: $\exists\Theta : t_1\Theta = t_2$, where Θ is an abstract substitution.

Example 8 (Pre-ordering on $ADom_P$)
- $a_4 \succcurlyeq f(a_2, g_1)$
- $g_1 \succcurlyeq f(g_2, g_3)$
- $f(h(a_1), g_1, a_1) \succcurlyeq f(h(a_3), h(g_2), a_3)$
- $f(h(a_1), g_1, a_1) \succcurlyeq f(h(k(a_2)), g_2, k(a_2))$

Proposition 5 (Pre-ordering in $ADom_P$)
$\succcurlyeq$ is a pre-ordering on $ADom_P$.

Proof.
Reflexivity of $\succcurlyeq$:
Obviously, for $t \in ADom_P$ and $\Theta_{Id} = \{a_i/a_i \mid a_i \in AVar(t)\} \cup \{g_j/g_j \mid g_j \in AVar(t)\}$, $t\Theta_{Id} = t$.
Transitivity of $\succcurlyeq$:
For two abstract substitutions, Θ_1 and Θ_2, their composition, $\Theta_2 \circ \Theta_1$, can be defined in the same way as composition of concrete substitutions. For each $t \in ADom_P$, it holds that $t(\Theta_2 \circ \Theta_1) = (t\Theta_1)\Theta_2$. Then, it is clear that, if $t_2 = t_1\Theta_1$ and $t_3 = t_2\Theta_2$, then $t_3 = (t_1\Theta_1)\Theta_2 = t_1(\Theta_2 \circ \Theta_1)$. □

Note that for any $t \in ADom_P$ and Θ an abstract substitution: $\gamma(t\Theta) \subseteq \gamma(t)$, because the abstract substitution only replaces abstract variables by potentially more instantiated abstract terms. Therefore, if $t_1 \succcurlyeq t_2$, then $\gamma(t_1) \supseteq \gamma(t_2)$.

Obviously, $\succcurlyeq$ is not anti-symmetric (e.g. both $a_1 \succcurlyeq a_2$ and $a_2 \succcurlyeq a_1$ hold) and therefore not a partial order. We use the standard approach for associating an equivalence relation and a partial order to the pre-order (see e.g. Baader and Nipkow [1999]).

Definition 12 (Equivalence and partial order on $ADom_P$)
Let t_1, $t_2 \in ADom_P$. t_1 and t_2 are equivalent, denoted $t_1 \approx t_2$, if $t_1 \succcurlyeq t_2$ and $t_2 \succcurlyeq t_1$.

The relation $\approx$ defines an equivalence relation on $ADom_P$. We denote the quotient set as $ADom_{P/\approx}$. For any $t \in ADom_P$, we denote by $\bar{t}$ its equivalence class in $ADom_{P/\approx}$. Let $\overline{t_1}$, $\overline{t_2} \in ADom_{P/\approx}$. $\overline{t_1} \leq \overline{t_2}$ if $t_1 \preccurlyeq t_2$, for some representatives t_1 of $\overline{t_1}$ and t_2 of $\overline{t_2}$. $\leq$ is well-defined and defines a partial order on $ADom_{P/\approx}$.

Note that $ADom_{P/\approx}$ merely formalizes the intuition that we have been using all along, that indices of a_i and g_j symbols in elements of $ADom_P$ are only relevant up to index renaming.

Next, we define how elements of the concrete domain are abstracted. We will not yet define the abstraction function in general, for any subset of Dom_P, but only on individual elements (singleton subsets) of Dom_P. We therefore refer to it as a pre-abstraction function.

Definition 13 (Pre-abstraction function)
A pre-abstraction function, $\alpha_1 : Dom_P \to ADom_P$, is a one-to-one function, mapping elements of Con_P to $\{g_i \mid i \in \mathbb{N}_0\}$ and mapping elements of Var_P to $\{a_i \mid i \in \mathbb{N}_0\}$ and such that $\forall t = f(t_1, \ldots, t_n) \in Dom_P$: $\alpha_1(t) = f(\alpha_1(t_1), \ldots, \alpha_1(t_n))$.

In what follows, by α_1, we denote a fixed pre-abstraction function. We also define $\overline{\alpha_1} : Dom_P \to ADom_{P/\approx}$ as $\overline{\alpha_1}(t) = \overline{(\alpha_1(t))}$.

$\overline{\alpha_1}$ may be seen as a "cleaner" formalization of the abstraction function, as it is independent of the selected α_1. Nevertheless, we will continue our formalization on the basis of representatives of equivalence classes, unless there is cause for confusion.

We now move towards the formalization of Abstract resolve. Every Horn clause, $h \leftarrow b_1, \ldots, b_m$, $m \in \mathbb{N}$, in P will be represented as $:\text{-}(h, b_1, \ldots, b_m)$.

To every Horn clause $C = :\text{-}(h, b_1, \ldots, b_m)$ in P, we can associate an abstract version of C: $AC = \alpha_1(:\text{-}(h_1, b_1, \ldots, b_n))$. Furthermore, we assume the availability of a renaming function on indices. More precisely, for each $t \in ADom_P$, there is a function $Rename(t) : ADom_P \rightarrow ADom_P$, such that for each $s \in ADom_P$, $Rename(t)(s)$ is an index renaming of all a_i and g_j occurring in s, such that all the abstract variables in $Rename(t)(s)$ are distinct from those in t.

Definition 14 (Abstract resolve)
Let $ConAt = \wedge(A_1, \ldots, A_i, \ldots, A_n) \in AConAtom_P$ and $AC = :\text{-}(H, B_1, \ldots, B_m)$ be an abstraction of Horn clause C of P, such that A_i and H have the same predicate symbol. Let $Rename(ConAt)(:\text{-}(H, B_1, \ldots, B_m))$ be $:\text{-}(H', B'_1, \ldots, B'_m)$. If Abstract unify for (A_i, H') terminates successfully and if the final mgu corresponds to $\Theta_{mgu}(A_i, H')$, then the abstract resolvent of ConAt and AC with selected literal A_i, $Aresolve(ConAt, AC, A_i)$, exists and is equal to $\wedge(A_1, \ldots, B'_1, \ldots, B'_m, \ldots, A_n)\Theta_{mgu}(A_i, H')$

Example 9 (Application of Abstract resolve)
We return to the construction of the second derivation tree for Permutation sort in Figure 2. Let us focus on the first Abstract resolve step, on the right-hand side of the tree. $ConAt = \wedge(perm(g_1, a_1), ord([g_2|a_1]))$. The applied Horn clause is $perm([X|Y], [U|V]) \leftarrow del(U, [X|Y], W), perm(W, V)$. The abstract clause associated to it, using the fixed α_1, AC, is $:\text{-}(perm([a_1|a_2], [a_3, a_4]), del(a_3, [a_1|a_2], a_5), perm(a_5, a_4))$. A possible renaming with respect to $ConAt$ could be $:\text{-}(perm([a_6|a_2], [a_3|a_4]), del(a_3, [a_6|a_2], a_5), perm(a_5, a_4)$. The selected atom in $ConAt$ is $perm(g_1, a_1)$. Then Abstract unify is applied to $perm(g_1, a_1)$ and $perm([a_6|a_2], [a_3|a_4])$. Algorithm 1 terminates successfully and with final set $mgu = \{a_6 = g_3, a_2 = g_4, a_1 = [a_3|a_4], g_1 = [g_3|g_4]\}$.

Thus $\Theta_{mgu} = \{a_6/g_3, a_2/g_4, a_1/[a_3|a_4], g_1/[g_3|g_4]\}$.

Then, $Aresolve(ConAt, AC, perm(g_1, a_1)) = \wedge(del(a_3, [g_3|g_4], a_5), perm(a_5, a_4), ord([g_2, a_3|a_4])$, which is a renaming of the abstract conjunction of the next node of Figure 2.

Proposition 6 (Correctness of Abstract resolve)
Let $ConAt \in AConAtom_P$, A_i an abstract atom in ConAt and AC an abstraction of a Horn Clause C of P, such that $Aresolve(ConAt, AC, A_i)$ exists. Let $conat \in \gamma(ConAt)$ and cA_i the atom in conat corresponding to A_i. Let the concrete resolvent of conat and C, with selected literal cA_i, be $cresolve(conat, C, cA_i) \in ConAtom_P$. If $cresolve(conat, C, cA_i)$ exists, then $cresolve(conat, C, cA_i) \in \gamma(Aresolve(ConAt, AC, A_i))$.

Proof. Let $ConAt = \wedge(A_1, \ldots, A_i, \ldots, A_n)$ and $Rename(ConAt)(:\text{-}(H, B_1, \ldots, B_m)) = :\text{-}(H', B'_1, \ldots, B'_m)$, so that $Aresolve(ConAt, AC, A_i) = \wedge(A_1, \ldots, B'_1, \ldots, B'_m, \ldots, A_n)\Theta_{mgu(A_i,H')}$. Let $conat = \wedge(cA_1, \ldots, cA_i, \ldots, cA_n)$ and the renamed clause of C be $:\text{-}(h', b'_1, \ldots, b'_m)$, so that $cresolve(conat, C, cA_i) = \wedge(cA_1, \ldots, b'_1, \ldots, b'_m, \ldots, cA_n)\delta_{mgu(cA_i,h')}$.

We need to prove: $cA_j\delta_{mgu(cA_i,h')} \in \gamma(A_j\Theta_{mgu(A_i,H')})$, for all $j = 1, \ldots, i-1, i+1, \ldots, n$, and $b'_j\delta_{mgu(cA_i,h')} \in \gamma(B'_j\Theta_{mgu(A_i,H')})$, for all $j = 1, \ldots, m$. It is given that $cA_j \in \gamma(A_j)$ and $b'_j \in \gamma(B'_j)$ for all relevant j. Therefore, the term structures of cA_j and A_j (and b'_j and B'_j) are the same: they differ only in the subterms of A_j (and B'_j) that are abstract variables.

Therefore, it suffices to prove for each abstract variable a_k (and g_l) occurring in some A_j (or B'_j), with corresponding concrete term ca_k (or cg_l) in cA_j (or b'_j) that:

$ca_k\delta_{mgu(cA_i,h')} \in \gamma(a_k\Theta_{mgu(A_i,H')})$

$cg_l\delta_{mgu(cA_i,h')} \in \gamma(g_l\Theta_{mgu(A_i,H')})$

There are two cases to consider. In the first case, a_k (or g_l) occurs in (A_i, H'). Then, the result follows directly from Proposition 4. In the second case, in which a_k (or g_l) does not occur in (A_i, H'), then $a_k\Theta_{mgu(A_i,H')} = a_k$ (and $g_l\Theta_{mgu(A_i,H')} = g_l$). Then, the result follows from the closedness under substitution of our abstract domain. □

4.3. Correctness of the approach

We are now ready to position our approach with respect to the main correctness theorem of the ACPD framework of Leuschel [2004].

First, Leuschel [2004] imposes two important conditions on the abstract domain: *downward closedness* and *the existence of concrete dominators*.
Downward closedness means that:

$$\forall A \in ADom_P, \forall t \in \gamma(A) : \{t\theta \mid \theta \text{ a substitution}\} \subseteq \gamma(A)$$

Clearly, our $ADom_P$ satisfies this property, by construction.
Existence of a concrete dominator means that:

$$\forall A \in ADom_P, \exists t \in Dom_P : \gamma(A) \subseteq \{t\theta \mid \theta \text{ a substitution}\}$$

So, for any abstract conjunction, A, there exists a concrete conjunction, t, so that all the concrete conjunctions represented by A are instances of this t. As an example, the condition excludes an abstract domain in which some $\gamma(A)$ contains both $p(x)$ and $\wedge(p(x), p(f(x))$: the length of all concrete conjunctions represented by A needs to be the same. As another example, the condition excludes an abstract domain in which some $\gamma(A)$ contains both $p(x)$ and $q(x)$: the concrete conjunctions represented by A must all consist of fixed sequence of atoms, having the same sequence of predicate symbols.

The reason for imposing this condition is that it ensures that the usual way of generating resolvents, borrowed from concrete partial deduction, can still be applied in the abstract case. Again, the existence of a concrete dominator is clearly satisfied in our $ADom_P$.

Next, the ACPD framework of Leuschel [2004] introduces direct generalizations of the key correctness notions of $\mathcal{A}$-*closedness* and *independence* from the (concrete) partial deduction framework of Lloyd and Shepherdson [1991]. The generalization of the former is referred to as $\mathcal{A}$-coveredness.

Concerning $\mathcal{A}$-coveredness, in our approach, as illustrated in Section 3, the process of developing abstract derivation trees is continued until $\mathcal{A}$-coveredness is achieved.

Independence means that for every A_i and $A_j \in \mathcal{A}$, $\gamma(A_i) \cap \gamma(A_j) = \emptyset$. Independence can always be achieved by appropriate renaming of predicates. Leuschel [2004] requires that these renamings respect some basic correctness properties and refers to such renamings as those producing *admissible renamed variants*. The renaming we apply in our approach, such as the ones illustrated in Section 3, produce admissible renamed variants.

Next, we recall the definition of an abstract partial deduction of P with respect to $\mathcal{A}$ from Leuschel [2004]. The definition is based on two functions, *aunfold*/2 and *aresolve*/2.

Informally, for a program P and an abstract conjunction A, $aunfold(P, A)$ is a finite set of concrete resultants computed from P, such that these resultants are "totally correct" for all conjunctions in $\gamma(A)$: no computed answers are lost or added. Definition 5.3 from Leuschel [2004] provides some fairly technical conditions, formalizing the above. We will not restate these conditions here, because Proposition 5.6 in Leuschel [2004] provides and proves a simple sufficient condition under which the conditions on *aunfold* are fulfilled. The latter condition will be sufficient for our purpose.

In our approach, for $A \in \mathcal{A}$, $aunfold(P, A)$ is the set of all concrete resultants computed in the second phase of our analysis, for the concrete conjunction $c(A)$, using the same derivation steps as in the abstract tree for A. The correctness of our *aunfold*/2 is ensured by the following proposition:

Proposition 7 (Leuschel [2004], Proposition 5.6)
Let $\boldsymbol{Q}$ be an abstract conjunction and let Q be a concrete dominator for $\boldsymbol{Q}$. Let τ be a SLD-tree for $P \cup \{\leftarrow Q\}$ and let $R \subseteq resultants(\tau)$ be a set of resultants such that for all resultants $Q\theta \leftarrow B \in (resultants(\tau) \backslash R)$ we have that no instance of $Q\theta$ is in $\gamma(\boldsymbol{Q})$. Then, $aunfold(P, \boldsymbol{Q}) = R$ satisfies Definition 5.3 from Leuschel [2004].

In our approach, for $A \in \mathcal{A}$, $c(A)$ is a concrete dominator for A. In fact, $c(A)$ is a maximally specific concrete dominator for A. By construction, the derivations considered for $c(A)$ in our second analysis mimic the derivations in the abstract tree for A. Some of the derivations in the considered concrete tree for $c(A)$ could fail and not give rise to a resultant C_i. This is not a problem: the abstract derivations are only a safe approximation. On the other hand, additional concrete derivations could exist for $c(A)$, which we are not considering in the concrete tree because they have no counterpart in the abstract tree. They correspond to $resultants(\tau) \backslash R$ in Proposition 7. This is not a problem either: by Proposition 6, *Aresolve*/3 safely approximates all the concrete derivations for conjunctions in $\gamma(A)$. If additional ones exist for $c(A)$, say with resolvent $c(A)\theta \leftarrow B$, then $c(A)\theta \notin \gamma(A)$.

Thus, Proposition 7 holds for our approach and we are within the framework of Leuschel [2004].

The second function introduced in Leuschel [2004], *aresolve*/2, maps each abstract conjunction $A \in \mathcal{A}$

and resultant $C_i \in \mathit{aunfold}(P, A)$ to an abstract conjunction $\mathit{aresolve}(A, C_i)$, which safely approximates all possible resolvent goals that can occur after resolving an element of $\gamma(A)$ with C_i.

In our approach, by construction, for every $C_i \in \mathit{aunfold}(P, A)$, there is a branch in the corresponding abstract tree for A, consisting of a finite sequence of $\mathit{Aresolve}/3$ applications, mimicking the concrete derivation from which C_i was obtained. By Proposition 6, $\mathit{Aresolve}/3$ safely approximates every concrete derivation step. Thus, the abstract conjunction in the leaf of that branch safely approximates all possible concrete resolvent goals for elements of $\gamma(A)$ with C_i. Therefore, our approach also respects the condition on $\mathit{aresolve}/2$ of Leuschel [2004], Definition 5.4.

We can now recall the definition of an abstract partial deduction.

Definition 15 (Abstract partial deduction, Leuschel [2004], Definition 7.6)
Let $\mathcal{A}$ be an $\mathcal{A}$-covered set of abstract conjunctions. We then define an abstract partial deduction of P wrt $\mathcal{A}$ to be the set of clauses:

$\{\rho_{\boldsymbol{A}}(H) \leftarrow \rho_{\mathcal{A},\boldsymbol{A}'}(B) \mid H \leftarrow B \in \mathit{aunfold}(P, \boldsymbol{A}) \wedge \boldsymbol{A}' = \mathit{aresolve}(\boldsymbol{A}, H \leftarrow B) \wedge \boldsymbol{A} \in \mathcal{A}\}$

The functions $\rho_{\mathbf{A}}$ and $\rho_{\mathcal{A},\mathbf{A}'}$ are renamings producing admissible renamed variants (Leuschel [2004], Definitions 7.4, 7.5 and 8.1). Our program transformation, then, is defined as Definition 15, using our own functions, corresponding to $\mathit{aunfold}/2$ and $\mathit{aresolve}/2$.

Finally, as our approach satisfies all the conditions of Leuschel [2004], the following correctness result holds for it:

Theorem 1 (*Correctness of the abstract partial deduction, Leuschel [2004], Theorem 8.2*)
Let P' be an abstract partial deduction of P wrt an $\mathcal{A}$-covered set of abstract conjunctions $\mathcal{A}$ and let Q' be an admissible renamed variant of Q wrt $\mathcal{A}$.
Then

1. *If $P \cup \{\leftarrow Q\}$ has an SLD-refutation with computed answer θ then $P' \cup \{\leftarrow Q'\}$ has an SLD-refutation with computed answer θ' such that $Q\theta$ is a variant of $Q'\theta'$.*
2. *If $P' \cup \{\leftarrow Q'\}$ has an SLD-refutation with computed answer θ' then $P \cup \{\leftarrow Q\}$ has an SLD-refutation with computed answer θ such that $Q\theta$ is a variant of $Q'\theta'$.*
3. *If $P' \cup \{\leftarrow Q'\}$ has a finitely-failed SLD-tree then so does $P \cup \{\leftarrow Q\}$.*

As a concluding observation, we want to motivate why we have not used the functions $\mathit{aunfold}/2$ and $\mathit{aresolve}/2$ from the start to introduce our instance of the ACPD framework. These functions are well-suited to formalize and reason about abstract partial deduction, but they are less suited to develop a new instance of the framework. In particular, $\mathit{aresolve}/2$ is defined in terms of $\mathit{aunfold}/2$, which suggests that an application would first generate concrete resultants and then develop abstract derivation trees associated with these resultants. It is more natural — certainly in our application — to first compute the abstract derivation trees and then produce the resultants on the basis of these.

4.4. Control and widening

Apart from the correctness issues, ACPD requires control. In Leuschel [2004], control issues are related to the vast amount of work done on the topic in partial deduction.

In our context, there are two concerns with respect to control. A first concern relates to the decisions regarding the selection rule and the full abstract interpretation of abstract atoms, both dealt with by the oracle.

A practical way to (indirectly) specify the selection rule is to provide delay declarations and to order tests before generators in programs. This is how coroutines are typically enforced in Prolog programs, including the program in Listing 1. The delay declarations express when a test is sufficiently instantiated to be unfolded. Using such declarations, we can ensure that a system takes the right decisions on which subgoal to unfold next.

For the second functionality of the oracle, the full abstract interpretation of *some* abstract atoms, a user of our approach and system needs to specify which abstract atoms are not involved in the coroutine and can simply be solved. The $\mathit{del}/3$ predicate of Section 3 is of this type. Note that it is not always sufficient to specify this information on the level of predicates, it is on the level of abstract atoms. This is also illustrated in the permutation sort example: $\mathit{ord}([g_1])$ and $\mathit{ord}([g_1, g_2])$ atoms can be fully evaluated, while $\mathit{ord}([g_1, g_2|a_1])$ atoms take part in the coroutine.

A third issue in the control is the termination of the analysis. Ensuring a terminating analysis for partial deduction has been the focus of much research in the past. Quite a number of proposals have been made, including several by the second author of the current paper (e.g. Leuschel et al. [1998], Martens et al. [1994], Leuschel and Martens [1996]). We have not yet studied the termination issues of our approach in much detail, but we expect that the techniques in Leuschel [2004] will be applicable.

In any case, especially because our abstract domain is infinitely large, a widening operator is required to ensure that a finite covered set $\mathcal{A}$ can always be reached. We provide a function to compute the widening of abstract terms and atoms.

In the definition of a widening function below, $+_b$ denotes modulo-2 addition.

Definition 16 (Widening)
A widening function $\vee : ATerm_P \times ATerm_P \cup AAtom_P \times AAtom_P \rightarrow ATerm_P \cup AAtom_P$ *is a one-to-one function, defined as:*
for $A_0, A_1 \in ATerm_P \cup AAtom_P \cup AConAtom_P$*:*

- *If* A_i*, for some* $i \in \{0,1\}$*, is of the form* a_j*, then* $A_0 \vee A_1 \in \{a_k \mid k \in \mathbb{N}_0\}$.
- *Else, if* A_i*, for some* $i \in \{0,1\}$*, is of the form* g_j*:*
  - *If* $A_{i+_b 1}$ *does not contain any symbol* a_l*,* $A_0 \vee A_1 \in \{g_k \mid k \in \mathbb{N}_0\}$.
  - *Otherwise,* $A_0 \vee A_1 \in \{a_k \mid k \in \mathbb{N}_0\}$.
- *Else, if* $A_i = f(t_1, \ldots, t_n)$ *and* $A_{i+_b 1} = h(s_1, \ldots, s_m)$*, for some* $i \in \{0,1\}$*:*
  - *If* $f \neq h$ *or* $n \neq m$
    - *If no* A_i*,* $i \in \{0,1\}$*, contains any symbol* a_l*:* $A_0 \vee A_1 \in \{g_k \mid k \in \mathbb{N}_0\}$.
    - *Otherwise* $A_0 \vee A_1 \in \{a_k \mid k \in \mathbb{N}_0\}$.
  - *Otherwise,* $A_0 \vee A_1 = f(t_1 \vee s_1, \ldots, t_n \vee s_n)$

For $A_0, A_1 \in AConAtom_P$, we consider this as a special case of the third case above, with $f = g = \wedge$.
In what follows, by $\vee$, we denote a fixed widening function.

Example 10 (Widening)
We provide an illustration for each of the cases expressed in Definition 16.

$\vee(a_2, p(f(g_1), a_1)) = a_3$

$\vee(g_1, h(g_2, f(g_3))) = g_4$

$\vee(g_2, f(a_1)) = a_2$

$\vee(f(g_1), h(g_2, g_3)) = g_5$

$\vee(f(g_1, g_2), h(a_1)) = a_4$

$\vee(f(a_1, g_1, h(g_1, a_1)), f(a_2, g_3, h(g_4, a_2))) = f(a_5, g_6, h(g_7, a_5))$

Given two abstract terms or atoms, it is tedious, but not hard to verify, that the widening operator, $\vee/2$, computes a minimally larger element of $ATerm_P$ or $AAtom_P$, with respect to $\succcurlyeq /2$. All these minimally larger elements are equivalent under $\approx /2$.

Similarly, given two abstract terms or atoms, abstract unify computes a maximally smaller element of $ATerm_P$ or $AAtom_P$, with respect to $\preccurlyeq /2$. Again, these are equivalent under $\approx /2$. As such, we establish that $(ATerm_{P/\approx}, \geq)$ and $(AAtom_{P/\approx}, \geq)$ are lattices.

Using the widening operator $\vee/2$, we can now extend Definition 13 to an abstraction function on 2^{Dom_P}.

Definition 17 (Abstraction function)
Let $S \subseteq Term_P$ *or* $S \subseteq Atom_P$ *or* $S \subseteq AConAtom_P$.
Then the abstraction function on S *is defined as:* $\alpha(S) = \bigvee_{s \in S} \alpha_1(s)$.

5. A More Complex Example, Introducing the *multi* Abstraction

In Section 3 we have shown that ACPD is indeed sufficient to formally revisit CC for a simple example. However, for more complex examples, ACPD still lacks expressivity. Consider the following prime number generator.

Example 11 (Prime numbers)

```
primes(N,P) ← integers(2,I),sift(I,P),len(P,N).
integers(N,[]).
integers(N,[N|I]) ← M is N+1, integers(M,I).
sift([N|Is],[N|Ps]) ← filter(N,Is,F), sift(F,Ps).
sift([],[]).
divides(N,M) ← 0 is M mod N.
not_divide(N,M) ← M mod N > 0.
filter(N,[M|I],F) ← divides(N,M), filter(N,I,F).
filter(N,[M|I],[M|F]) ← not_divide(N,M), filter(N,I,F).
filter(N,[],[]).
len([],0).
len([H|T],N) ← M is N - 1, len(T,M).
```

The program is intended to be called with a goal $primes(N, P)$, with N a positive integer and P a free variable. The *integers*/2 predicate generates growing lists of consecutive integer numbers. *filter*/3 represents the removal of all multiples of a single integer N from a list. *sift*/2 recursively filters out multiples of an initial list element which is prime. The SWI-Prolog program in Listing 3 provides a concrete implementation of this control flow. The non-logical *freeze/2* predicate and reordering of atoms are required.

```
primes(N,Primes) :-
  freeze(Primes,len(Primes,N)),
  freeze(I,sift(I,Primes)),
  integers(2,I).

integers(N,[]).
integers(N,[N|I]) :- M is N+1, integers(M,I).

sift([N|Ints],[N|Primes]) :-
  freeze(Ints,filter(N,Ints,F)),
  freeze(F,sift(F,Primes)).
sift([],[]).

divides(N,M) :- 0 is M mod N.
does_not_divide(N,M) :- M mod N > 0.

filter(N,[M|I],F) :- divides(N,M), freeze(I,filter(N,I,F)).
filter(N,[M|I],[M|F]) :- does_not_divide(N,M), freeze(I,filter(N,I,F)).
filter(N,[],[]).

len([],0).
len([H|T],N) :- M is N - 1, freeze(T,len(T,M)).
```

Listing 3: SWI-Prolog implementation of the prime sieve

We only present the most relevant parts of the abstract partial deduction. The top level goal for the abstract analysis is $primes(g_1, a_1)$, so that the initial set $\mathcal{A}$ is $\{primes(g_1, a_1)\}$. A first abstract derivation tree describes the initialization for the computation. Its rightmost branch leads to the leaf: $\wedge(integers(g_3, a_3), filter(g_2, a_3, a_6), sift(a_6, a_5), len(a_5, g_4))$, which is added to $\mathcal{A}$.

Next, we construct an abstract derivation tree for the latter conjunction. For the case where, in a concrete execution, a number passes through the filter, we add the following conjunction to $\mathcal{A}$:
$\wedge(integers(g_5, a_{10}), filter(g_2, a_{10}, a_7), filter(g_3, a_7, a_8), sift(a_8, a_9), len(a_9, g_6))$.

At this point it becomes clear that an analysis following only the steps shown in Section 3 will not terminate. The two abstract conjunctions, most recently added to $\mathcal{A}$, are identical – up to renaming of a_i's and g_j's – except that the latter conjunction contains two atoms $filter(g_k, a_i, a_j)$, instead of just one. A further analysis, building additional derivation trees, will result in the construction of continuously growing conjunctions, with continuously increasing numbers of $filter/3$ atoms.

We could solve this by cutting the goal into two smaller conjunctions and adding these to $\mathcal{A}$. However, all these atoms are generators or testers in the coroutine and depend on each other. By splitting the conjunction, we would no longer be able to analyze the coroutine. We therefore avoid splitting conjunctions if the evaluation of their atoms is interleaved.

Recall that ACPD requires that, for any abstract conjunction of atoms, $acon \in AConAtom_P$, there exists a concrete dominator, $con \in ConAtom_P$, such that: for all $con_i \in \gamma(acon)$: con_i is an instance of con. In practice, this means that an abstract conjunction is not allowed to represent a set of concrete conjunctions whose elements have a distinct number of conjuncts. However, in order to solve the problem observed in our example, we need the ability to represent a set of conjunctions, with a growing number of atoms, by an abstract atom. Therefore, we need to extend ACPD.

We will extend our abstract domain and introduce a new abstraction, $multi/4$, which makes it possible to represent growing conjunctions, with a number of copies of a single abstract atom.

As the new abstraction function is fairly complex, let us first introduce the intuition. Assume that we have a number of abstract conjunctions in which there is one abstract atom with an increasing number of occurrences, while all other abstract atoms have a fixed number of occurrences. We can rename the indices of a_i and g_i variables occurring in the atoms whose frequency increases, such that the indices in abstract atoms **not** involved in the growing subconjunction are fixed.

For Example 11, the following conjunctions are renamed variants of abstract conjunctions we aim to generalize:

- $\wedge(integers(g_3, a_3), filter(g_1, a_3, a_5), sift(a_5, a_4), len(a_4, g_4))$ (3)
- $\wedge(integers(g_3, a_3), filter(g_1, a_3, a_6), filter(g_2, a_6, a_5), sift(a_5, a_4), len(a_4, g_4))$
- $\wedge(integers(g_3, a_3), filter(g_1, a_3, a_6), filter(g_2, a_6, a_7), filter(g_5, a_7, a_5), sift(a_5, a_4), len(a_4, g_4))$

We wish to introduce a new syntactic construct to represent the growing subconjunction of $filter(g_i, a_j, a_k)$ atoms. This construct should have the following features:

- It should specify the abstract atom repeated in the subconjunction ($filter(g_i, a_j, a_k)$).
- It should allow us to reconstruct all the aliasing on abstract variables, both between abstract atoms inside the subconjunction and between abstract atoms of the subconjunction with abstract atoms outside the subconjunction.

In our example, the new abstraction that we propose to represent the subconjunction is:

$$\begin{aligned} &multi(filter(g_{Id(A),\mu,1}, a_{Id(A),\mu,3}, a_{Id(A),\mu,6}),\\ &\qquad \wedge\,(a_{Id(A),1,3} = a_3),\\ &\qquad \wedge\,(a_{Id(A),\mu+1,3} = a_{Id(A),\mu,6}),\\ &\qquad \wedge\,(a_{Id(A),\mathcal{L},6} = a_5)) \end{aligned} \tag{4}$$

Here, $A = filter(g_1, a_3, a_6)$.

The first argument in the abstraction represents the abstract atom ($filter(g_1, a_3, a_6)$) which is repeated in the subconjunction, but also introduces a new naming scheme for the indices of a_i and g_j variables. This naming scheme has three components:

- $Id(A)$ is an identifier for $A = filter(g_1, a_3, a_6)$
- The second component is a symbolic index. It can have four possible values: 1, $\mathcal{L}$, μ and $\mu + 1$. 1 and $\mathcal{L}$ respectively refer to the first and last atom of the conjunction. μ and $\mu + 1$ refer to a pair of consecutive atoms of the conjunction.

Most often, we will use the symbolic index to represent the shared pattern of the abstracted atoms. We will then use the Greek letter μ. For instance, $filter(g_{Id(A),\mu,1}, a_{Id(A),\mu,3}, a_{Id(A),\mu,6})$ represents the pattern of the abstract atoms in the subconjunction. We can use the symbol μ to refer to a_j symbols of consecutive $filter(g_1, a_3, a_6)$ atoms in the sequence, like in an equation $a_{Id(A),\mu+1,3} = a_{Id(A),\mu,6}$.
We also allow instantiating the symbolic index to a number or a symbol representing a specific number. For instance, with $a_{Id(A),1,3}$, we can refer to the a_3 variable in the **first** $filter(g_1, a_3, a_6)$ atom in the sequence. With $a_{Id(A),\mathcal{L},6}$, we can refer to the a_6 variable in the **last** $filter(g_1, a_3, a_6)$ atom in the sequence. Here, $\mathcal{L}$ is still a symbol, but it represents one specific (unknown) number related to the sequence.

- The last argument is the index of the a_i or g_j in the $filter(g_1, a_3, a_6)$ atom.

The second, third and fourth arguments of the abstraction (4) are conjunctions of constraints on the (newly named) a_i's. In our example, each of these conjunctions has only one conjunct. The second argument, $\wedge(a_{Id(A),1,3} = a_3)$, expresses the aliasing between the first $filter(g_1, a_3, a_6)$ atom in the sequence and the abstract atoms not contained in the subconjunction. The third argument, $\wedge(a_{Id(A),\mu+1,3} = a_{Id(A),\mu,6})$, expresses the aliasing between two consecutive $filter(g_1, a_3, a_6)$ atoms in the sequence. The fourth argument, $\wedge(a_{Id(A),\mathcal{L},6} = a_6)$, expresses the aliasing between the last $filter(g_1, a_3, a_6)$ in the sequence and the abstract atoms not contained in the subconjunction.

We formally introduce the parameterized naming scheme for a_i and g_i variables and apply this to abstract atoms.

Definition 18 (Parameterized naming and parameterized abstract atom)
Let $A \in AAtom_P$. By $Id(A)$, we denote a unique identifier associated with A.

Let $a_j \in AVar_P, j \in \mathbb{N}_0$, such that a_j occurs in A, then the parameterized naming of a_j is the symbol $a_{Id(A),\mu,j}$.

Let $g_k \in AVar_P, k \in \mathbb{N}_0$, such that g_k occurs in A, then the parameterized naming of g_k is the symbol $g_{Id(A),\mu,k}$.

Let $A \in AAtom_P$. The parameterized atom for A, $p(A)$, is obtained by replacing every a_j and g_k occurring in A by their parameterized namings, $a_{Id(A),\mu,j}$ and $g_{Id(A),\mu,k}$.

The new abstraction *multi*/4 will depend on the abstract conjunction in which it occurs. This conjunction may contain abstract variables, a_j and g_k. It may also contain parameterized namings of abstract variables, $a_{Id(A),\mu,j}$ and $g_{Id(A),\mu,k}$. This is due to the fact that a *multi*/4 abstraction will typically contain parameterized namings and that an abstract conjunction will be allowed to contain multiple *multi*/4 abstractions. Therefore, the abstract conjunction containing one *multi*/4 abstraction may also contain another *multi*/4 abstraction.

Given an abstract conjunction C, we denote $a(C) = \{a_j \in AVar_P \mid a_j \text{ occurs in } C\}$. We also denote $g(C) = \{g_k \in AVar_P \mid g_k \text{ occurs in } C\}$. Furthermore, we denote $pa(C) = \{a_{Id(A),1,j} \mid a_{Id(A),1,j} \text{ occurs in } C\} \cup \{a_{Id(A),\mathcal{L},j} \mid a_{Id(A),\mathcal{L},j} \text{ occurs in } C\}$. We also denote $pg(C) = \{g_{Id(A),1,k} \mid g_{Id(A),1,k} \text{ occurs in } C\} \cup \{g_{Id(A),\mathcal{L},k} \mid g_{Id(A),\mathcal{L},k} \text{ occurs in } C\}$.

Example 12
For Example 11, referring to the three conjunctions (3) we aim to represent, the abstract conjunction providing the context for subconjunctions of $filter(g_1, a_3, a_6)$ atoms is $C = \wedge(integers(g_3, a_3), sift(a_5, a_4), len(a_4, g_4))$. $a(C) = \{a_3, a_4, a_5\}$, $g(C) = \{g_3, g_4\}$, $pa(C) = pg(C) = \emptyset$.

Definition 19 (Multi abstraction)
Let C be an abstract conjunction. A multi abstraction is a construct of the form multi($p(A)$, First, Consecutive, Last), where:
- $p(A)$ is the parameterized atom for some $A \in AAtom_P$.
- First is a conjunction of equalities $a_{Id(A),1,j} = b_j$, where $b_j \in a(C) \cup pa(C) \cup g(C) \cup pg(C)$ and all left-hand sides of the equalities are distinct, and equalities $g_{Id(A),1,k} = b_k$, where $b_k \in g(C) \cup pg(C)$ and all left-hand sides of the equalities are distinct.
- Consecutive is a conjunction of equalities $a_{Id(A),\mu+1,j} = a_{Id(A),\mu,j'}$, where $j, j' \in \mathbb{N}_0$ and all left-hand sides of the equalities are distinct, and equalities $g_{Id(A),\mu+1,k} = g_{Id(A),\mu,k'}$, where $k, k' \in \mathbb{N}_0$ and all left-hand sides of the equalities are distinct.
- Last is a conjunction of equalities $a_{Id(A),\mathcal{L},j} = b_j$, where $b_j \in a(C) \cup pa(C) \cup g(C) \cup pg(C)$ and all left-hand sides of the equalities are distinct, and equalities $g_{Id(A),\mathcal{L},k} = b_k$, where $b_k \in g(C) \cup pg(C)$ and all left-hand sides of the equalities are distinct.

Example 13 (Multi abstraction)
We return to the primes example and consider the three abstract conjunctions:

- $\wedge(integers(g_3, a_3), filter(g_1, a_3, a_5), sift(a_5, a_4), len(a_4, g_4))$
- $\wedge(integers(g_3, a_3), filter(g_1, a_3, a_6), filter(g_2, a_6, a_5), sift(a_5, a_4), len(a_4, g_4))$
- $\wedge(integers(g_3, a_3), filter(g_1, a_3, a_6), filter(g_2, a_6, a_7), filter(g_5, a_7, a_5), sift(a_5, a_4), len(a_4, g_4))$

Let $A = filter(g_1, a_3, a_6)$ and $C = \wedge(integers(g_3, a_3), sift(a_5, a_4), len(a_4, g_4))$. Then, the three abstract conjunctions can be generalized, using the *multi*/4 abstraction, to: $\wedge(integers(g_3, a_3), multi(filter(g_{Id(A),\mu,1}, a_{Id(A),\mu,3}, a_{Id(A),\mu,6}), \wedge(a_{Id(A),1,3} = a_3), \wedge(a_{Id(A),\mu+1,3} = a_{Id(A),\mu,6}), \wedge(a_{Id(A),\mathcal{L},6} = a_5)), sift(a_5, a_4), len(a_4, g_4))$.

We can now define the new abstract domain. Given an abstract conjunction *Con*, with a subconjunction *Sub*, by $Con - Sub$, we refer to the conjunction obtained by removing *Sub* from *Con*.

Definition 20 (Abstract-multi domain)
Let $AAtomMulti_P = AAtom_P \cup \{multi(p(A), First, Consecutive, Last) \mid A \in AAtom_P$ *and First, Consecutive and Last are conjunctions of equalities* $a_{Id(A),\mu,j} = b_j$ *or* $g_{Id(A),\mu,k} = b_k$*, with* b_j, b_k *abstract variables or parameterized abstract variables* $\}$.

Let $AConAtomMulti_P$ *be the set of all conjunctions Con of elements of* $AAtomMulti_P$*, such that each of its* $multi(p(A), First, Consecutive, Last)$ *conjuncts respects the conditions of Definition 19, with respect to the set* $C = Con - \{multi(p(A), First, Consecutive, Last)\}$.

The Abstract-multi domain, $ADomMulti_P$*, is* $AVar_P \cup ATerm_P \cup AAtom_P \cup AConAtomMulti_P$.

Abstract conjunctions containing *multi*/4 abstractions represent infinitely many abstract conjunctions without the *multi*/4 abstraction. In the example, these contain either one or multiple $filter(g_k, a_i, a_j)$ atoms.

In what follows, we will omit the $Id(A)$ subscript in the parameterized namings $a_{Id(A),\mu,j}$ and $g_{Id(A),\mu,k}$ and just refer to $a_{\mu,j}$ and $g_{\mu,k}$ instead. The $Id(A)$ subscript is only relevant for abstract conjunctions containing multiple *multi*/4 abstractions, a case which we will not consider for the moment.

In order to describe the abstract conjunctions represented by an abstract conjunction containing a *multi*/4 abstraction, we need the ability to map parameterized namings back to ordinary a_j and g_k variables. This requires the following concepts.

Definition 21 (Concrete index assignment mapping)
Let $n \in \mathbb{N}_0$*. The concrete index assignment mapping,* $R(\mu, n)$*, is a mapping defined on any syntactic construct, S, containing parameterized namings* $a_{\mu,j}$*,* $a_{\mu+1,j}$*,* $g_{\mu,k}$ *or* $g_{\mu+1,k}$*.* $R(\mu, n)$ *replaces every occurrence of a parameterized naming* $a_{\mu,j}$*,* $a_{\mu+1,j}$*,* $g_{\mu,k}$ *and* $g_{\mu+1,k}$ *in S by the parameterized naming* $a_{n,j}$*,* $a_{n+1,j}$*,* $g_{n,k}$ *and* $g_{n+1,k}$*, respectively.*

Note that, while $\mu + 1$ is a symbol, $n + 1$ is a number.

Example 14 (Concrete index assignment mapping)
$R(\mu, 1)(filter(g_{\mu,1}, a_{\mu,3}, a_{\mu,6})) = filter(g_{1,1}, a_{1,3}, a_{1,6})$.

Definition 22 (Double-index mapping)
Let i be any symbolic index. The double-index mapping, ψ*, is a mapping defined on any syntactic construct, S, containing parameterized namings* $a_{i,j}$ *or* $g_{i,k}$*.* ψ *replaces every occurrence of a parameterized naming* $a_{i,j}$ *and* $g_{i,k}$ *in S by* a_{i_j}*, respectively* g_{i_k} *where* i_j *and* i_k *denote fresh elements of* $\mathbb{N}_0$*, not occurring as the index of any abstract variable yet.*

Example 15 (Double-index mapping)
$\psi(filter(g_{\mu,1}, a_{\mu,3}, a_{\mu,6})) = filter(g_{\mu_1}, a_{\mu_3}, a_{\mu_6})$, with μ_1, μ_3, μ_6 fresh elements of $\mathbb{N}_0$.

Definition 23 (Substitution corresponding to equality constraints)
Let i be any symbolic index. Let Constraint be a conjunction of equality constraints of the form $a_{i,j} = b_j$ *and* $g_{i,k} = b_k$*, with* $a_{i,j}$ *and* $g_{i,k}$ *parameterized namings, and such that all left-hand sides of equalities are mutually distinct. The substitution corresponding to Constraint is the substitution* $\Theta_{Constraint} = \{\psi(a_{i,j})/\psi(b_j) \mid a_{i,j} = b_j \in Constraint\} \cup \{\psi(g_{i,k})/\psi(b_k) \mid g_{i,k} = b_k \in Constraint\}$.

Note that this definition is meant to deal with the conjunctions of equalities in the *First*, *Consecutive* and *Last* arguments of the *multi*/4 abstraction. The *Last* argument requires some extra attention. Sometimes, during analysis, we will actually know the value of $\mathcal{L}$. To avoid confusion with the symbol $\mathcal{L}$, we denote this value as l.

Example 16 (Substitutions corresponding to equality constraints)
For the conjunctions of equality constraints within the *multi*/4 abstraction in Example 13, and after assigning a concrete value to μ, say m, the corresponding substitutions are: $\Theta_{First} = \{a_{1_3}/a_3\}$, $\Theta_{Consecutive} = \{a_{(m+1)_3}/a_{m_6}\}$, $\Theta_{Last} = \{a_{l_6}/a_5\}$.

With these notions, we can now describe the abstract conjunctions represented by a *multi*/4 abstraction.

Definition 24 (Abstract conjunctions represented by *multi*/4)
Let $l \in \mathbb{N}_0$. The abstract conjunctions represented by multi$(p(A), First, Consecutive, Last)$ are:

- $\psi(R(\mu,1)(p(A)))\Theta_{First} \circ \Theta_{R(\mathcal{L},1)(Last)}$
- $\psi(R(\mu,1)(p(A)))\Theta_{First} \wedge \psi(R(\mu,2)(p(A)))\Theta_{R(\mu,1)(Consecutive)} \wedge \ldots \wedge \psi(R(\mu,l)(p(A)))\Theta_{R(\mu,l-1)(Consecutive)} \circ \Theta_{Last}$, *with $l > 1$.*

Example 17 (Abstract conjunctions represented by *multi*/4)
For the *multi*/4 abstraction in Example 13, *multi*$(filter(g_{\mu,1}, a_{\mu,3}, a_{\mu,6}), \wedge(a_{1,3} = a_3), \wedge(a_{\mu+1,3} = a_{\mu,6}), \wedge(a_{\mathcal{L},6} = a_5))$, the represented abstract conjunctions are:

- $filter(g_{1_1}, a_3, a_5)$
- $filter(g_{1_1}, a_3, a_{1_6}) \wedge filter(g_{2_1}, a_{1_6}, a_{2_3}) \wedge \ldots \wedge filter(g_{l_1}, a_{(l-1)_6}, a_5), l > 1$.

Note that, by applying Definition 24 and by using the concretization function γ on the resulting elements of $ADom_P$, we have now defined the concretization of all elements of $ADomMulti_P$.

Next, we need to define the abstract unfolding of a *multi*/4 abstraction. Unfolding a *multi*/4 abstraction makes a case split. Either the *multi*/4 abstraction represents only one abstract atom, or it represents more than one. In both cases the abstract bindings with the surrounding abstract conjunction and, in the latter case, the abstract bindings between consecutive atoms, need to be respected.

Definition 25 (Abstract unfold of *multi*/4)
Abstract unfold of multi produces a branching in the abstract derivation tree. An abstract atom multi$(p(A),$ $First, Consecutive, Last)$ is replaced in one branch by $\psi(R(\mu,1)(p(A)))\Theta_{First} \circ \Theta_{R(\mathcal{L},1)(Last)}$ and in a second branch by $\psi(R(\mu,1)(p(A)))\Theta_{First} \wedge multi(p(A), NewFirst, Consecutive, Last)$, where $NewFirst = \wedge\{a_{1,j} = a_{1_{j'}} \mid a_{(\mu+1),j} = a_{\mu,j'} \in Consecutive\} \bigwedge \wedge\{g_{1,k} = g_{1_{k'}} \mid g_{(\mu+1),k} = g_{\mu,k'} \in Consecutive\}$

Example 18 (Abstract unfold of *multi*/4)
Again returning to Example 13, abstract unfold of *multi*$(filter(g_{\mu,1}, a_{\mu,3}, a_{\mu,6}), \wedge(a_{1,3} = a_3), \wedge(a_{\mu+1,3} = a_{\mu,6}),$ $\wedge(a_{\mathcal{L},6} = a_5))$ produces both $filter(g_{1_1}, a_3, a_5)$ and $filter(g_{1_1}, a_3, a_{1_6}) \wedge multi(filter(g_{\mu,1}, a_{\mu,3}, a_{\mu,6}), \wedge(a_{1,3} = a_{1_6}), \wedge(a_{\mu+1,3} = a_{\mu,6}), \wedge(a_{\mathcal{L},6} = a_5))$.

A few comments on this definition are in order. First, the definition of *NewFirst* may seem strange, because both sides of the equalities have a "1" index. However, note that on the left-hand side of the equality, it is in a parameterized naming, e.g. $a_{1,j}$, referring to the first atom represented by the *multi*/4, while on the right-hand side, it is in an abstract variable $a_{1_{j'}}$, referring to an atom that was just moved outside of the *multi*/4. Second, it is important to remember that the abstract variables a_{1_j} and g_{1_k} are produced by $\psi(a_{1,j})$ and $\psi(g_{1,k})$ calls and that their indices 1_j and 1_k need to be fresh indices, not yet occurring with any a or g variables, respectively. This is particularly important in cases where we perform several abstract unfoldings of *multi*/4 in sequence. At each unfold, new fresh subscripts need to be introduced.

Finally, we need to define abstract generalization with *multi*/4, allowing us to replace conjunctions of identically instantiated and similarly aliased abstract atoms by a *multi* construct.

Definition 26 (Abstract generalization with *multi*/4)
Let $A \in AAtom_P$. Let $A_1, \ldots, A_n \in AAtom_P$ and let $\bigwedge_{m=1,\ldots,n} A_m$ occur in an abstract conjunction Con. Let $C = Con - (\wedge_{m=1,\ldots,n} A_m)$. Let $a(C)$, $g(C)$ and $pg(C)$, $pa(C)$ respectively be the abstract variables and the parameterized namings occurring in C. Let $r_m, m = 1, \ldots, n$, be renamings of A, such that $r_m(A) = A_m$.

$Gen(\bigwedge_{m=1,\ldots,n} A_m) = multi(p(A), First, Consecutive, Last)$ is the abstract generalization with multi/4 of $\bigwedge_{m=1,\ldots,n} A_m$ in C if:

- *for any $b_j \in a(C) \cup pa(C), a_{1,j} = b_j \in First$ if and only if $r_1(a_j) = b_j$*
- *for any $b_k \in g(C) \cup pg(C), g_{1,k} = b_k \in First$ if and only if $r_1(g_k) = b_k$*

- $a_{\mu+1,j} = a_{\mu,j'} \in$ *Consecutive if and only if for all* $p = 1, \ldots, n-1 : r_{p+1}(a_j) = r_p(a_{j'})$
- $g_{\mu+1,k} = g_{\mu,k'} \in$ *Consecutive if and only if for all* $p = 1, \ldots, n-1 : r_{p+1}(g_k) = r_p(g_{k'})$
- *for any* $b_j \in a(C) \cup pa(C)$, $a_{\mathcal{L},j} = b_j \in$ *Last if and only if* $r_n(a_j) = b_j$
- *for any* $b_k \in g(C) \cup pg(C)$, $g_{\mathcal{L},k} = b_k \in$ *Last if and only if* $r_n(g_k) = b_k$

We can extend Definition 26 to generalizations $Gen(\wedge_{m=1,\ldots,k} A_m \wedge multi(p(A), First, Consecutive, Last)) = multi(p(A), First', Consecutive, Last)$ and $Gen(multi(p(A), First, Consecutive, Last), \wedge \bigwedge_{m=1,\ldots,\mathcal{L}} A_m) = multi(p(A), First, Consecutive, Last')$. We omit the details for these generalizations. We will illustrate abstract generalization with *multi*/4 in our running example below.

Let us return to the prime numbers example. Observing the growing number of *filter*/3 atoms in the second abstract conjunction of Example 13 (w.r.t. the conjunction already present in $\mathcal{A}$), we perform the generalization. Recall that $C = \wedge(integers(g_3, a_3), sift(a_5, a_4), len(a_4, g_4))$, so that $a(C) = \{a_3, a_4, a_5\}$ and $g(C) = \{g_3, g_4\}$. $Gen(\wedge(filter(g_1, a_3, a_6), filter(g_2, a_6, a_5))) = multi(filter(g_{1,\mu,1}, a_{1,\mu,3}, a_{1,\mu,6}), \wedge(a_{1,1,1} = a_3), \wedge(a_{1,\mu+1,3} = a_{1,\mu,6}), \wedge(a_{1,\mathcal{L},6} = a_5))$. Here, we include the $Id(A)$ again, because we will have multiple *multi*/4 abstractions. We arbitrarily select $Id(A)$ to be 1.

Then we construct a new abstract derivation tree for this conjunction, including – among others – an abstract unfold of *multi*/4 and abstract generalizations with *multi*/4. In Figure 5, we show this abstract tree.

After abstract unfolding of $integers(g_3, a_3)$, the tree contains an abstract unfolding of $multi(filter(g_{1,\mu,1}, a_{1,\mu,3}, a_{1,\mu,6}), \wedge(a_{1,1,3} = [g_3|a_7]), \wedge(a_{1,\mu+1,6} = a_{1,\mu,3}), \wedge(a_{1,\mathcal{L},6} = a_5))$. This unfolding can lead to one instance of *filter*/3 or several. A full evaluation of $does\_not\_divide(g_6, g_3)$ leads to a new generalization which produces a renaming of the root of this tree.

Eventually, the analysis ends up with a final set $\mathcal{A}$:

$$
\begin{aligned}
&\{ \wedge (primes(g_3, a_3)),\\
&\quad \wedge (integers(g_3, a_3), multi(filter(g_{1,\mu,1}, a_{1,\mu,3}, a_{1,\mu,6}), \wedge(a_{1,1,3} = a_3),\\
&\qquad \wedge (a_{1,\mu+1,3} = a_{1,\mu,6}), \wedge(a_{1,\mathcal{L},6} = a_5)), sift(a_5, a_4), len(a_4, g_4)),\\
&\quad \wedge (multi(filter(g_{1,\mu,1}, a_{1,\mu,3}, a_{1,\mu,6}), \wedge(a_{1,1,3} = g_5), \wedge(a_{1,\mu+1,3} = a_{1,\mu,6}),\\
&\qquad \wedge (a_{1,\mathcal{L},6} = a_5)), sift(a_5, a_4), len(a_4, g_4)),\\
&\quad \wedge (integers(g_5, a_7), multi(filter(g_{2,\mu,1}, a_{2,\mu,3}, a_{2,\mu,6}), \wedge(a_{2,1,3} = a_7),\\
&\qquad \wedge (a_{2,\mu+1,3} = a_{2,\mu,6}), \wedge(a_{2,\mathcal{L},6} = a_8)), multi(filter(g_{1,\mu,1}, a_{1,\mu,3}, a_{1,\mu,6}),\\
&\qquad \wedge (a_{1,1,3} = [g_6|a_{2,\mathcal{L},6}]), \wedge(a_{1,\mu+1,3} = a_{1,\mu,6}), \wedge(a_{1,\mathcal{L},6} = a_5)), sift(a_5, a_4),\\
&\qquad len(a_4, g_4))\}
\end{aligned}
$$

All non-empty leaves in the abstract derivation trees for these atoms are (renamings of) elements of $\mathcal{A}$. This shows $\mathcal{A}$-coveredness and the abstract phase of the analysis terminates.

Similar to what was observed for permutation sort in Section 3, we still need an extra analysis to collect the concrete bindings, so that the resultants can be generated. Special care is required for the *multi*/4 abstraction. There are three issues: how to represent *multi*/4 in the concrete domain, how to deal with the concrete counterparts of abstract generalization with *multi*/4 and abstract unfolding of *multi*/4.

Definition 5, in Section 3, defined the concrete counterparts of the conjunctions in $\mathcal{A}$. We extend it to $multi(A)$:

Definition 27 (Concrete conjunction for $multi(A, First, Consecutive, Last)$)
Let $A \in AAtom_P$, *then* $c(multi(p(A), First, Consecutive, Last)) = multi([c(A)|T])$, *with* T *a fresh variable.*

It may seem strange that in the concrete analysis phase we omit the three arguments *First*, *Consecutive* and *Last*. These arguments are needed in the abstract analysis to correctly capture the data flow and to correctly model the unfolding under the coroutining selection rule. In the concrete analysis phase, as we are completely mimicking the unfolding in the corresponding abstract trees, we are still performing the correct selection. Moreover, the only point of the concrete analysis phase is to collect the concrete bindings produced by unfolding the concrete clauses. The extra arguments are not needed for this purpose.

Example 19 (Concrete conjunction for $multi(A, First, Consecutive, Last)$)
$c(multi(filter(g_{1,\mu,1}, a_{1,\mu,3}, a_{1,\mu,6}), \wedge(a_{1,1,3} = a_3), \wedge(a_{1,\mu+1,3} = a_{1,\mu,6}), \wedge(a_{1,\mathcal{L},6} = a_5))) =$
$multi([filter(X, I1, F1)|T])$

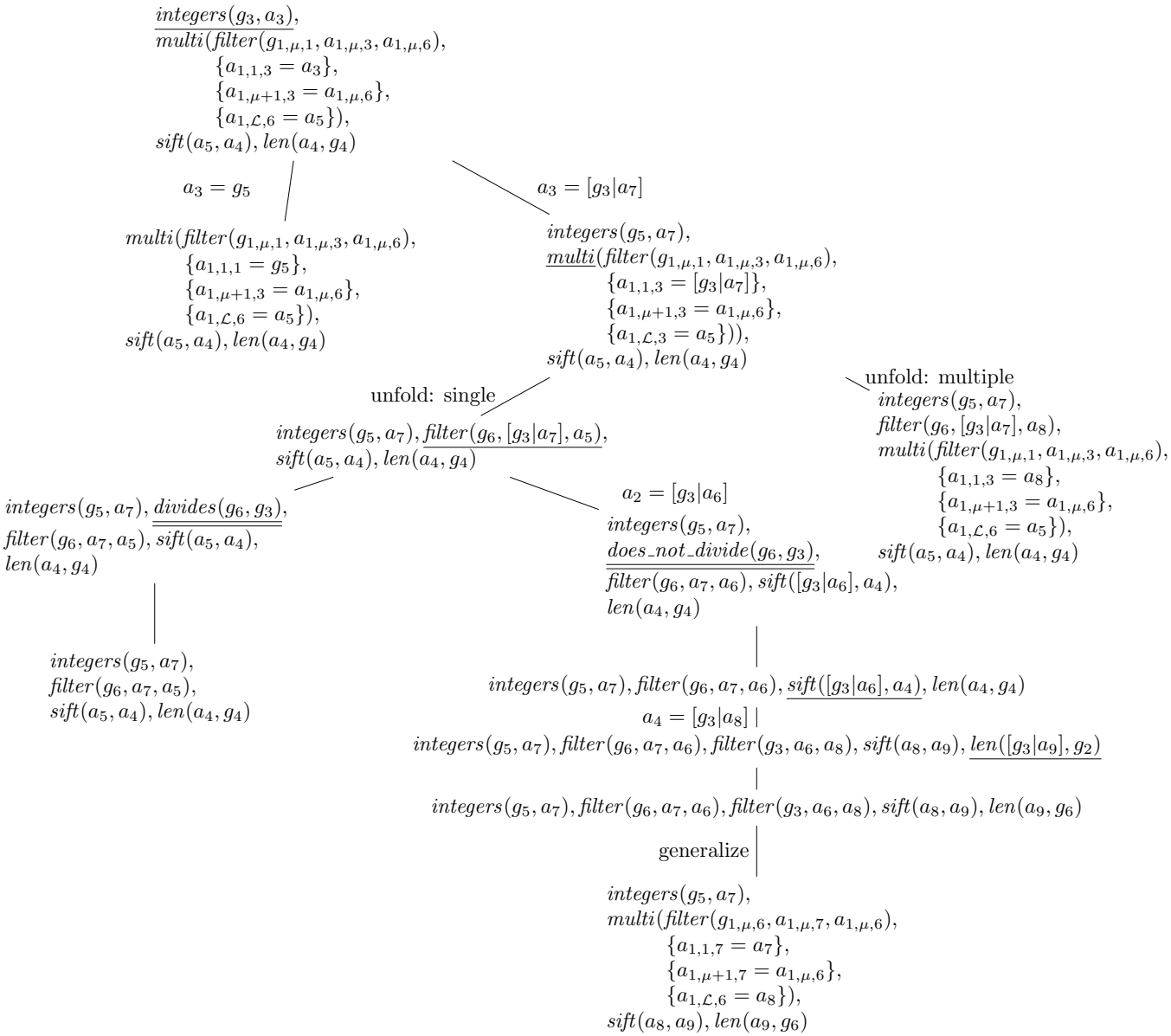

Figure 5. Use of the *multi*/4 abstraction in the prime sieve

For the abstract generalization with *multi*/4, we define the concrete counterpart as follows.

Definition 28 (Concrete generalization)
Let $A \in AAtom$.

- *If the abstract generalization with multi/4 is of the type $Gen(\bigwedge_{i=1,\ldots,n} A) = multi(A, First, Consecutive, Last)$, then the corresponding node in the concrete derivation contains $c(\bigwedge_{i=1,\ldots,n} A)$. The concrete generalization is defined as $ConGen(c(\bigwedge_{i=1,\ldots,n} A)) = multi(c([A, \ldots, A]))$, with n members in the list.*
- *If the abstract generalization with multi/4 is of the type $Gen((\bigwedge_{i=1,\ldots,n} A) \wedge multi(A, First, Consecutive, Last)) = multi(A, First', Consecutive, Last)$, then the corresponding node in the concrete derivation contains $c(\bigwedge_{i=1,\ldots,n} A) \wedge multi(List)$, where List is a list of at least one $c(A)$. The concrete generalization*

is defined as $ConGen(c(\bigwedge_{i=1,\ldots,n} A) \wedge multi(List)) = multi([c(A), \ldots, c(A)|List])$ *with* n *new members added to* $List$.

- *The third case,* $Gen(multi(A, First, Consecutive, Last) \wedge (\bigwedge_{i=1,\ldots,n} A)) = multi(A, First, Consecutive, Last')$*, is treated similarly to the previous case, but the concrete atoms are appended to the existing list.*

Example 20 (Concrete generalization)
Let $integers(A, B), filter(C, B, D), filter(E, D, F), sift(F, G), len(G, H)$ occur in a concrete conjunction in a concrete derivation tree, where abstract generalization with $multi/4$ is performed on the corresponding abstract conjunction. Then, as a next step in the concrete derivation tree, this conjunction is replaced by $integers(A, B), multi([filter(C, B, D), filter(E, D, F)]), sift(F, G), len(G, H)$.

Note that this "generalization" actually does not generalize anything. It only brings the information in a form that can be generalized.

The actual generalization happens implicitly in the move to the construction of the next concrete derivation tree. If our conjunction is a leaf of the concrete derivation tree, then the corresponding abstract conjunction is added to the set $\mathcal{A}$. Let $\wedge(integers(g_3, a_3), multi(filter(g_{1,\mu,1}, a_{1,\mu,3}, a_{1,\mu,6}), \wedge(a_{1,1,3} = a_3), \wedge(a_{1,\mu+1,3} = a_{1,\mu,6}), \wedge(a_{1,\mathcal{L},6} = a_5)), sift(a_5, a_4), len(a_4, g_4))$, for instance, be the corresponding abstract conjunction that is added to $\mathcal{A}$. Then, a new concrete tree is built for a concrete conjunction corresponding to this abstract one.

In this example, the root of that concrete tree is:

$\wedge(integers(A, B), multi([filter(C, B, D)|T]), sift(E, F), len(F, G))$

Finally, we still need to define the counterpart of abstract unfold of $multi/4$ in the concrete tree. To do this, we add the following definition of $multi/1$ to the original program P.

```
multi([H]) ← H.
multi([H|T]) ← H, multi(T).
```

It should be clear that concrete unfolding of concrete $multi/1$ atoms with the above definition for $multi/1$ gives us the desired counterpart of the case split performed in abstract unfold of $multi/1$ if we apply concrete bindings analogous to abstract bindings used in the abstract unfold.

With the concepts above, we construct a concrete derivation tree, mimicking the steps in the abstract derivation tree – but over the concrete domain – for every conjunction in the set $\mathcal{A}$. Collecting all the resultants from these concrete trees, we get the transformed program. A working Prolog program can be found in Annex [2014].

6. Implementation and discussion

To demonstrate that our approach can be automated, we have implemented a prototype system. This system is available at Annex [2016]. It implements the basic operations that we discuss in the paper: abstraction of clauses, abstract unify, abstract resolve, for an abstract clause and an abstract conjunction from $\mathcal{A}$ and verification of $\mathcal{A}$-coveredness.

In terms of the decisions by the oracle, the system requires manually specified instructions by a user. For the selection rule it uses delay declarations on the test-predicates. For the calls for which complete abstract interpretation is required, it requires declarations from the user of the abstract atom that needs to be completely solved, together with the output atom that is obtained after solving the atom in abstract interpretation. Of course, in future work, we intend to link the system to an abstract interpretation system, for instance implementing the framework of Bruynooghe [1991], to produce the output abstractions automatically.

For the moment, the control component ensuring widening and termination, has not been automated. None of the examples dealt with so far have required widening. In fact, examples that do need widening on the level of the arguments of the generators and tests may not benefit from the approach. If we need to widen the abstract patterns on which the control rule is based, then the system will no longer be able to capture the control rule. We have analyzed the effect of widening on some examples. What happens in such cases is that the approach still synthesizes a correct program, but the resulting program does not implement the coroutining selection rule, because the computations in the abstract resolves did not capture the desired selection rule. Reconsider Figure 1: if, instead of adding $perm(g_5, a_3), ord([g_4|a_3])$ to $\mathcal{A}$, we widen

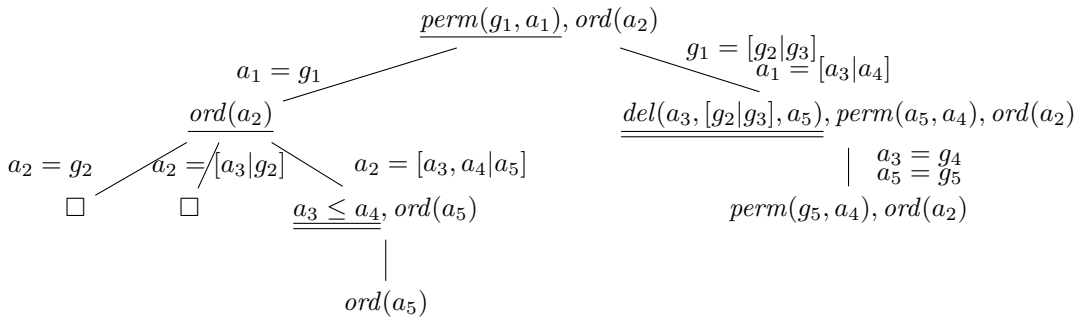

Figure 6. Widening causes the control to be lost

it to $perm(g_5, a_3), ord(a_4)$, we obtain a closed analysis which does not interleave a generator and a tester. This analysis is shown in Figure 6.

The system also requires the user to specify which conjunctions of growing abstract conjunctions should be generalized into a *multi*/4. On the basis of these specifications, the system produces the operations of abstract unfold of *multi* and generalization. As such, the system can deal with more complex examples, such as the prime numbers example.

We have experimented with the system on a number of examples, including permutation sort, graph coloring, n-queens, lucky numbers and prime numbers. The code of these examples is presented at Annex [2015]. The analysis and synthesis was successful for all these examples, providing us with additional evidence that our theory and the proposed technique in general are sound.

We also performed some experiments with examples that are outside the scope of our current suggestion for the *multi*/4 abstraction. In an attempt to find the limitations of the approach, we have considered examples with unboundedly growing conjunctions that can not be represented as a growing linear sequence of atoms. A typical example is the sameleaves program.

Example 21 (Sameleaves)

```
sameleaves(T1,T2) ← collect(T1,T1L), collect(T2,T2L), eq(T1L,T2L).
eq([],[]).
eq([H|T1],[H|T2]) ← eq(T1,T2).
collect(node(X),[X]).
collect(tree(L,R),C) ← collect(L,CL), collect(R,CR), append(CL,CR,C).
append([],L,L).
append([H|T],L,[H|TR]) ← append(T,L,TR).
```

In a coroutining execution of this program, goals are growing unboundedly, but *multi*/4 is not expressive enough to capture the unbounded growth as a mere sequence of identically linked atoms. The example requires that the building blocks within the *multi*/4 are not atoms, but conjunctions of atoms.

Given the abstract conjunctions, occurring in a coroutining execution of sameleaves:

- $collect(g_1, a_1), \underline{collect(g_2, a_2), append(a_1, a_2, a_3)}, collect(g_3, a_4), eq(a_3, a_4)$
- $collect(g_1, a_1), \underline{collect(g_2, a_2), append(a_1, a_2, a_3), collect(g_3, a_4), append(a_3, a_4, a_5)}, collect(g_4, a_6), eq(a_5, a_6)$
- $collect(g_1, a_1), \underline{collect(g_2, a_2), append(a_1, a_2, a_3), collect(g_3, a_4), append(a_3, a_4, a_5), collect(g_4, a_6), append(a_5, a_6, a_7)}, collect(g_5, a_8), eq(a_7, a_8)$

there is the subconjunction $collect(g_2, a_2), append(a_1, a_2, a_3)$ which has a growing number of occurrences in a fixed sequential pattern $collect(g_i, a_i), append(a_j, a_i, a_k)$. If we replace the role of the atom in the *multi*/4 abstraction by a conjunction of atoms, then our *multi*-extension is again capable of capturing this behavior. Our system has been extended to capture this extension and can also analyze the sameleaves example.

As a result, we now believe that the *multi*/4 extension on ACPD can be made powerful enough to deal with any type of coroutine, provided that we allow nested *multi*/4's. A key issue in future work is to prove that any coroutine can indeed be captured by such a further extension of the *multi*-abstraction.

A second key issue is the soundness of the *multi*-abstraction. The abstraction does not fit within the ACPD framework, as it explicitly violates Leuschel's requirement that, for every abstract conjunction, there is a concrete dominator. We will relax this requirement and establish correctness results for all operations involving the abstraction.

We have also performed experiments on the runtime efficiency of our transformed programs. As most of our transformations convert a naive generate and test program into a coroutining version, the results clearly show improved execution times. However, when compared to the non-compiled coroutining versions (for instance, using a delay mechanism to implement the coroutine), our transformations are not more efficient. This is due to highly efficient implementations of delay mechanisms and the fact that our transformations introduce significantly more pattern matching into the program.

However, note that, unlike for the CC transformation, the main objective of our work is no longer efficiency. It is the ability to provide a better, formal understanding of the computation flow of coroutines. If we can completely capture the computation flow of coroutines in a formal analysis, many analysis techniques that were developed in the past for left-to-right execution rules for logic programming, can now be ported to coroutining selection rules.

7. Conclusions

In this paper, we have presented an approach to formally analyze the computations, for Logic Programs, performed under coroutining selection rules, and to compile such computations into new programs. On the basis of an example, we have shown that simple coroutines, in which the execution of a single, atomic generator is interleaved with a single, atomic tester, can be successfully analyzed and compiled within the framework of ACPD (Leuschel [2004]). These "simple" coroutines essentially correspond to the *strongly regular* logic programs of Vidal [2011], based on Hruza and Stepánek [2004].

To achieve this, we defined an expressive abstract domain, capturing modes, types and aliasing. This abstract domain is a refinement of the abstract domain that we introduced in De Schreye et al. [2014], in the sense that it now allows expressing the equality of "g"-variables, in addition to aliasing of "a"-variables. We provide a full formalization of the approach, including Abstract unify, Abstract resolve, the order on the abstract domain, widening and the abstraction function.

Because the approach – for simple coroutines – fits fully within the ACPD framework, it inherits the correctness results from ACPD. In particular, $\mathcal{A}$-coveredness and independence guarantee the completeness and correctness of the analysis. In addition, the transformation preserves all computed answers (in both directions) and finite failure of the transformed program implies finite failure of the original.

We have proposed an extension to our abstract domain: the *multi*/4-abstraction. A *multi*/4 atom can represent (sets of) conjunctions of one or more concrete atoms. We have defined abstract unfold and abstract generalisation operations for this abstraction. We have shown, in an example, that this abstraction and these operations allow us to extend ACPD, enabling it to perform a complete analysis, and to compile the more complex coroutines.

On a more general level, our work provides a new, rational reconstruction of the CC-transformation (Bruynooghe et al. [1986]), avoiding ad hoc features of the CC approach. In addition, the work presents a new application for ACPD.

We have developed a prototype implementation of the approach and tested it on a number of examples. The prototype validates our formal results and shows that it achieves the expected results in practice. Some components of the system, such as the global control of the partial deduction and the detection of the need for *multi*/4 generalization have not been automated as yet. They require manual intervention of the user. These are challenges for future work.

As a rule, coroutining improves the efficiency of declarative programs by testing partial solutions as quickly as possible. In addition, a program may become more flexible when the transformation is applied. For instance, a generate-and-test based program for the graph coloring problem which was transformed in the course of this research was originally meant to be called with a ground list of nations and a list of free variables of the correct length. A transformed variant of this program can be run in the same way, but the top-level predicate can also be called with a ground list of nations and a free variable. This is because SLD resolution sends the original program down an infinite branch of the search tree. The transformed program checks results earlier and, as a result, infers that both top-level arguments must be lists of the same size. In this scenario, compiling control transforms an infinite computation into a finite one.

The CC-transformation raised challenges for a number of researchers and a range of compediting transformation and synthesis techniques. A first reformulation of the CC-transformation was proposed in the context of the "programs-as-proofs" paradigm, in Wiggins [1990]. It was shown that CC-transformations, to a limited extent, could be formalized in a proof-theoretic program synthesis context.

In Boulanger et al. [1993], CC-transformation was revisited on the basis of a combination of abstract interpretation and constraint processing. This improved the formalization of the technique, but it did not clarify the relation with partial deduction.

The seminal survey paper on Unfold/Fold transformation, Pettorossi and Proietti [1994], showed that basic CC-transformations are well in the scope of Unfold/Fold transformation. In later works (e.g. Pettorossi and Proietti [2002]), the same authors introduced list-introduction into the Unfold/Fold framework, whose function is very similar to that of the *multi*/4 abstraction in our approach. Also related to our work are Puebla et al. [1997], providing alternative transformations to improve the efficiency of dynamic scheduling, and Vidal [2011] and Vidal [2012], which also provide a hybrid form of partial deduction, combining abstract and concrete levels.

There are a number of issues that are open for future research. First, we aim to investigate the generality of the *multi*/4 abstraction. Based on some additional experiments, we believe that a further extension of the *multi*/4 abstraction that makes it possible to represent linear sequences of *conjunctions* of abstract atoms, instead of just linear sequences of abstract atoms, will be able to cover all practical examples. We also want to revisit the ACPD framework, in order to extend it to the new abstraction we aim to support. This will involve a new formalization of ACPD, capable of supporting analysis and compilation of coroutines in full generality. This will also formally establish the correctness results for the more general cases, such as the one presented in Section 5. Obviously, we also want to extend our implementation of these and to show that the analysis and compilation can be fully automated. Finally, the applicability of our analysis in the presence of non-logical features of Prolog, such as the "cut" operator and negation, is subject to further investigation.

Acknowledgements

Vincent Nys is supported by FWO Flanders under research contract G088414N. The authors thank the anonymous reviewers for very constructive feedback on an earlier draft of the paper.

References

Annex. Extended concepts acpd. `http://perswww.kuleuven.be/~u0055408/generalization-acpd.html`, 2014.

Annex. Examples of coroutining programs. `http://perswww.kuleuven.be/~u0055408/analyzed-examples-acpd.html`, 2015.

Annex. Implementation of ACPD. `http://perswww.kuleuven.be/~u0055408/implementation-acpd.html`, 2016.

Franz Baader and Tobias Nipkow. *Term rewriting and all that*. Cambridge university press, 1999.

Dmitri Boulanger, Maurice Bruynooghe, and Danny De Schreye. Compiling control revisited: A new approach based upon abstract interpretation for constraint logic programs. In *Proceedings of the 5th Workshop on Logic Programming Environments (LPE 1993), October 29-30, 1993, In conjunction with ILPS 1993, Vancouver, British Columbia, Canada*, pages 39–51, 1993.

Maurice Bruynooghe. A practical framework for the abstract interpretation of logic programs. *The Journal of Logic Programming*, 10(2):91–124, 1991.

Maurice Bruynooghe, Danny De Schreye, and Bruno Krekels. Compiling control. In *Proceedings of the 1986 Symposium on Logic Programming*, pages 70–77, Salt Lake City, 1986. IEEE Society Press.

Maurice Bruynooghe, Danny De Schreye, and Bruno Krekels. Compiling control. *The Journal of Logic Programming*, 6(1):135–162, 1989.

Emanuele De Angelis, Fabio Fioravanti, Alberto Pettorossi, and Maurizio Proietti. Proving correctness of imperative programs by linearizing constrained horn clauses. *TPLP*, 15(4-5):635–650, 2015. doi: 10.1017/S1471068415000289. URL `http://dx.doi.org/10.1017/S1471068415000289`.

Danny De Schreye, Robert Glück, Jesper Jørgensen, Michael Leuschel, Bern Martens, and Morten Heine Sørensen. Conjunctive partial deduction: Foundations, control, algorithms, and experiments. *The Journal of Logic Programming*, 41(2):231–277, 1999.

Danny De Schreye, Vincent Nys, and Colin Nicholson. Analysing and compiling coroutines with abstract conjunctive partial deduction. In *Logic-Based Program Synthesis and Transformation*, pages 21–38. Springer, 2014.

John P. Gallagher. Transforming logic programs by specialising interpreters. In *ECAI*, pages 313–326, 1986.

Jan Hruza and Petr Stepánek. Speedup of logic programs by binarization and partial deduction. *TPLP*, 4(3):355–380, 2004. doi: 10.1017/S147106840300190X. URL `http://dx.doi.org/10.1017/S147106840300190X`.

Henryk Jan Komorowski. *A specification of an abstract Prolog machine and its application to partial evaluation.* PhD thesis, Linköping University, 1981.

Michael Leuschel. A framework for the integration of partial evaluation and abstract interpretation of logic programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 26(3):413–463, 2004.

Michael Leuschel and Bern Martens. Global control for partial deduction through characteristic atoms and global trees. In *Partial Evaluation, International Seminar, Dagstuhl Castle, Germany, February 12-16, 1996, Selected Papers*, pages 263–283, 1996. URL `http://dx.doi.org/10.1007/3-540-61580-6_13`.

Michael Leuschel, Bern Martens, and Danny De Schreye. Controlling generalization and polyvariance in partial deduction of normal logic programs. *ACM Trans. Program. Lang. Syst.*, 20(1):208–258, 1998. URL `http://doi.acm.org/10.1145/271510.271525`.

John Lloyd. *Foundations of Logic Programming.* Berlin: Springer-Verlag, 1987.

John W. Lloyd and John C Shepherdson. Partial evaluation in logic programming. *The Journal of Logic Programming*, 11(3):217–242, 1991.

Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(2):258–282, 1982.

Bern Martens, Danny De Schreye, and Tamás Horváth. Sound and complete partial deduction with unfolding based on well-founded measures. *Theoretical Computer Science*, 122(1):97–117, 1994.

Alberto Pettorossi and Maurizio Proietti. Transformation of logic programs: Foundations and techniques. *The Journal of Logic Programming*, 19:261–320, 1994.

Alberto Pettorossi and Maurizio Proietti. The list introduction strategy for the derivation of logic programs. *Formal aspects of computing*, 13(3-5):233–251, 2002.

Germán Puebla, Maria J Garcia de la Banda, Kim Marriott, and Peter J Stuckey. Optimization of logic programs with dynamic scheduling. In *Logic Programming: Proceedings of the International Conference on Logic Programming*, volume 97, pages 93–107. MIT Press, 1997.

Germán Vidal. A hybrid approach to conjunctive partial evaluation of logic programs. In *Logic-Based Program Synthesis and Transformation*, pages 200–214. Springer-Verlag Berlin Heidelberg, 2011.

Germán Vidal. Annotation of logic programs for independent and-parallelism by partial evaluation. *Theory and Practice of Logic Programming*, 12(4-5):583–600, 2012.

Geraint A Wiggins. *The improvement of Prolog program efficiency by compiling control: A proof-theoretic view.* Department of Artificial Intelligence, University of Edinburgh, 1990.